TOWARDS SUCCESSFUL APPLICATION OF PHASE CHANGE MEMORIES: ADDRESSING CHALLENGES FROM WRITE OPERATIONS

by

Ping Zhou

B.S., Shanghai Jiao Tong University, 2001

M.S., Shanghai Jiao Tong University, 2004

Submitted to the Graduate Faculty of

the Swanson School of Engineering in partial fulfillment

of the requirements for the degree of

PhD, Computer Engineering

University of Pittsburgh

2012

UNIVERSITY OF PITTSBURGH

SWANSON SCHOOL OF ENGINEERING

This dissertation was presented

by

Ping Zhou

It was defended on

November 22nd, 2011

and approved by

Jun Yang, PhD, Associate Professor, Electrical and Computer Engineering Department

Youtao Zhang, PhD, Associate Professor, Department of Computer Science

Bruce Childers, PhD, Associate Professor, Department of Computer Science

Sangyeun Cho, PhD, Associate Professor, Department of Computer Science

Minhee Yun, PhD, Associate Professor, Electrical and Computer Engineering Department

Steven Levitan, PhD, John A. Jurenko Professor, Electrical and Computer Engineering Department

Dissertation Directors: Jun Yang, PhD, Associate Professor, Electrical and Computer Engineering Department,

Youtao Zhang, PhD, Associate Professor, Department of Computer Science

TOWARDS SUCCESSFUL APPLICATION OF PHASE CHANGE MEMORIES: ADDRESSING CHALLENGES FROM WRITE OPERATIONS

Ping Zhou, PhD

University of Pittsburgh, 2012

The emerging Phase Change Memory (PCM) technology is drawing increasing attention due to its advantages in non-volatility, byte-addressability and scalability. It is regarded as a promising candidate for future main memory. However, PCM's write operation has some limitations that pose challenges to its application in memory. The disadvantages include long write latency, high write power and limited write endurance.

In this thesis, I present my effort towards successful application of PCM memory. My research consists of several optimizing techniques at both the circuit and architecture level. First, at the circuit level, I propose Differential Write to remove unnecessary bit changes in PCM writes. This is not only beneficial to endurance but also to the energy and latency of writes. Second, I propose two memory scheduling enhancements (AWP and RAWP) for a non-blocking bank design. My memory scheduling enhancements can exploit intra-bank parallelism provided by non-blocking bank design, and achieve significant throughput improvement. Third, I propose Bit Level Power Budgeting (BPB), a fine-grained power budgeting technique that leverages the information from Differential Write to achieve even higher memory throughput under the same power budget. Fourth, I propose techniques to improve the QoS tuning ability of high-priority applications when running on PCM memory.

In summary, the techniques I propose effectively address the challenges of PCM's write operations. In addition, I present the experimental infrastructure in this work and my visions of potential future research topics, which could be helpful to other researchers in the area.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENTS

I want to express my sincere gratitude to my advisor, Professor Jun Yang, and my co-advisor, Professor Youtao Zhang. Professor Yang and Professor Zhang are extraordinary researchers that guide, inspire and care about their students. They taught me not only knowledge and skills, but also the way to conduct research, to think out-of-the-box and identify interesting problems from seemly common phenomenon. I'm touched by their diligence, inspiration, and the invaluable guidance and advice they gave me along the road. During the development of my thesis, I received extraordinary helps from Professor Yang and Professor Zhang, without which I could not have completed my dissertation. It has always been my honor and pleasure working with Professor Jun Yang and Professor Youtao Zhang.

I want to thank Professor Bruce Childers, Professor Sangyeun Cho, Professor Steven Levitan and Professor Minhee Yun, for their help on my research and my dissertation. They have provided me with discussions and comments which helped me a lot refining my manuscript. I want to thank Professor Yiran Chen for his advice on my research. I also appreciate all who have helped me by answering my questions, reviewing my papers, providing insightful comments and in many other ways. Your help has been very important to me in the development of this dissertation.

I want to thank my classmates, Bo Zhao, Yu Du, Lei Jiang, Lin Li, Yi Xu, for helping me whole-heartedly on many of my research projects. I received lots of helps from them on setting up the experimental environment, answering questions, discussing new ideas and looking up important parameters. Their help has been crucial to my achievements.

I want to thank my parents and my parents-in-law. They helped me through the most challenging days with their love, encouragement and unconditional support. Although they are not in Pittsburgh, I can feel their heart with me all the time.

Last but not least, I want to thank my wife Chunni Jin. I cannot imagine reaching this point without her sacrifice, care and support. No matter how challenging the day it is, Chunni Jin has always been on my side, helping me with her love, wisdom and courage. I cannot be grateful enough for everything she has done for me and our family. I am lucky to have her being my wife, and I wish her success in her professional career.

To my dear wife Chunni Jin, my dear daughter Sarah Zhou, my dear mother Lei Zhu and my dear father Yunsui Zhou

1.0 INTRODUCTION

The call for large and low power memory systems is continuing its pace especially in Chip Multiprocessors (CMP) due to the growing memory requirements of new applications and the increasing number of processing cores. As the technology enters the nanoscale regime, large DRAM-based main memory faces serious leakage and scalability limitations. For example, in a mid-range IBM eServer machine, 40% of the power was reported to be consumed by its main memory system [51]. Recent studies have shown that memory energy conservation should focus on leakage energy reduction since leakage grows with the memory capacity, and the main memory can dissipate as much leakage energy as dynamic energy [93]. DRAM is also facing serious problems scaling to 40nm or beyond, as it is constrained by the limitation in cell-bitline capacitance ratio [43, 61]. As a result, people have resorted to several non-volatile memory technologies as alternatives to conventional DRAM. Examples include NAND Flash, Phase Change Memory (PCM) and Spin-Transfer Torque RAM (STT-RAM).

Among these non-volatile memory technologies, NAND Flash a has very limited number of write/erase cycles: 10^5 rewrites [25] as opposed to 10^{16} for DRAM. NAND Flash also requires a block to be erased before writing into that block, which introduces considerably extra delay and energy. Moreover, NAND Flash can only be accessed in blocks and is not byte-addressable. Therefore, NAND flash has been proposed as a disk cache [9, 42] or a replacement for disks [97] where writes are relatively infrequent, and happen mostly in blocks.

Besides NAND Flash, two emerging memory technologies, PCM and STT-RAM, are regarded as promising candidates for next-generation memory technology. Both of them share the common advantages of non-volatility, byte-addressability and scalability, and are backed by key industry manufacturers such as Intel, ST-Microelectronics, Samsung, IBM and TDK [30, 38]. STT-RAM is faster than PCM, and has nearly the same endurance as DRAM (10^{15} [92] vs. 10^{16} rewrites). PCM, on the other hand, is much denser than STT-RAM. The cell area for DRAM, PCM and

STT-RAM are $6F^2$ [93], $5\sim8F^2$ [47], and $37\sim40F^2$ [20] respectively, where F is the feature size. PCM also has excellent scalability within current CMOS fabrication methodology [13,44,47,74,82]. PCM's attributes, along with the common advantages of emerging memory technologies, make it a promising candidate as part of main memory [48,80,103].

For a better understanding of the differences among memory technologies, I summarize their basic attributes in Table 1.

Table 1: Basic attributes of emerging memory technologies [20,27,39,47,52,67,92,93,103]

	Read Speed	Write Speed	Cell Area	Endurance	Byte-addressable
DRAM	20∼50ns	20∼50ns	$6F^2$	10^{15}	Yes
SRAM	∼2ns	∼2ns	$146F^2$	$10^{15} \sim 10^{16}$	Yes
NAND Flash	25us [a]	500us [b]	$5F^2$	$10^4 \sim 10^5$	No
STT-RAM	∼2ns	∼10ns	$37 \sim 40F^2$	10^{12}	Yes
PCM	**30∼50ns**	**∼1us**[c]	**5∼8**F^2	$\sim 10^8$	**Yes**

[a] Page read.

[b] Page write, block erasure required before write.

[c] Per-access (page write) time. For cell-access time (writing of single PCM cell), the number is typically 50∼150ns [48].

In this thesis, I assume PCM is used in main memory along with an optional DRAM buffer to form a hybrid memory system, as shown in Figure 1.

Despite PCM memory's low leakage and good scalability, however, a PCM-based memory system still has several issues to solve. The issues are mainly caused by PCM's write operation, which has several disadvantages:

Write endurance. PCM write is a thermally-driven process that involves heating and cooling (details of PCM operations are discussed in Chapter 2). Due to the repeated heat stress in this process, a PCM cell can be written for a limited number of times (typically $10^8 \sim 10^9$ times [25,39,99]). While this is better than the write endurance of NAND Flash (i.e., 10^5 times), it is worse than that of a DRAM cell (i.e., 10^{15}) times and is a big concern when PCM is used in main memory.

Write power. PCM write incurs high current injection (e.g., 0.6∼1mA [50]), resulting in high write power (e.g., 2.88∼4.8mW per bit [50]). Large numbers of concurrent PCM bit writes can

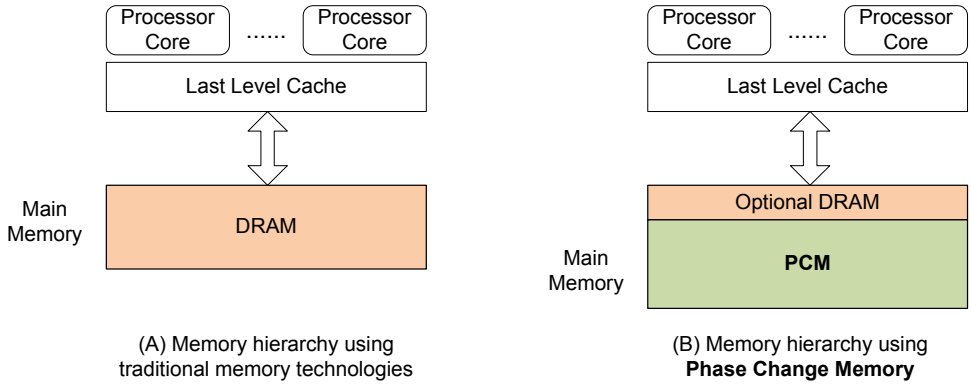


Figure 1: Potential application of PCM in memory hierarchy.

raise concern of transient write power and write current. For example, Kang et al. reported in their prototype that designing charge pumps for the write driver was challenging because they have to supply high current and sustain high voltage at same time [40]. Moreover, transient high current causes noise on the power line, and therefore the number of cells written in parallel has to be restricted [40]. Due to the same reason, Chung et al. used a scheme in which cell writes are skewed in time to avoid transient high current [15]. Hence, a practical PCM memory will typically have some limitation on the number of concurrent bit writes. This limitation is called a "power budget" in this thesis.

Write latency. PCM's write operation is much slower than its read operation. A typical latency of writing a single-level PCM cell is 90ns∼150ns [48, 103] (the write latency is worse in multi-level cells). Moreover, because of PCM's high write power and write current injection, writing a PCM memory line (e.g., 512-bit line) is usually completed in several *rounds*, with each round writing part of the line. This makes PCM's per-access write latency even longer. For example, Numonyx reported a 1us (1000ns) page write latency in their PCM prototype [67].

These disadvantages of PCM's write operation lead to several major challenges to the application of PCM memory. In addition to the lifetime issue caused by PCM's limited write endurance, PCM's long write latency also causes low memory throughput, as an active write request can block a bank and make subsequent requests wait for a long time. In order to improve throughput of PCM memory, more parallelism must be exploited. However, PCM's high write power makes this effort even more challenging, as practical PCM memory is typically designed with a limit on the number

of concurrent bit writes. When multiple applications are running concurrently, each application will experience longer read access latencies. For some high-priority applications, it is often desirable to be able to tune this "memory slowdown". Unfortunately, PCM's long write latency increases the interference among multiple concurrent applications and degrades this QoS tuning ability on high-priority applications.

Successful application of PCM memory demands techniques to address these challenges caused by PCM's disadvantageous write operations. In the next section, I will present an overview of my research work on this topic.

1.1 RESEARCH OVERVIEW

My research is a comprehensive effort towards successful application of PCM memory. It consists of the following building blocks and components at the circuit and architecture levels. An overview of my research is illustrated in Figure 2.

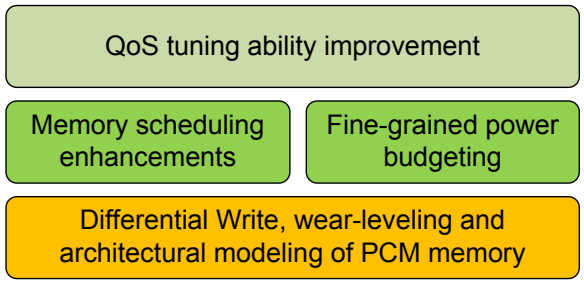


Figure 2: Overview of my research.

First, I propose a circuit-level technique Differential Write to remove unnecessary bit changes in PCM writes. Differential Write performs read and compare before each PCM write, and only writes the cells that are actually changed. By using Differential Write along with a set of simple wear-leveling techniques I propose, I extend PCM memory's lifetime significantly, addressing the major concern on PCM's write endurance. In addition, Differential Write helps reduce write power and opens new opportunities to the upper level (memory scheduling). The modeling and evaluation I did also demonstrates the energy efficiency of PCM-based memory.

Second, I propose my memory scheduling enhancements for a novel non-blocking bank design [105]. Although Differential Write is beneficial for write power and endurance, it does not help PCM's write latency a lot. A PCM write still takes a long time unless new data and old data are completely identical (i.e., every bit of the write is redundant). In conventional bank design, each logic bank serves one request at a time. This means an active PCM write can block subsequent requests for a long time, hurting both latency and throughput. Existing memory scheduling enhancements like write-cancelation and write-pausing [76] only help read latency but not throughput, as requests are still served in serial in each bank. In order to improve PCM memory's throughput, a non-blocking bank design is developed in which each logic bank can serve up to two reads and two writes simultaneously. However, throughput improvement is found to be limited if the memory scheduler is not aware of this new bank design. To exploit more throughput improvement from the non-blocking bank design, I propose two scheduling enhancements, Aggressive Write Precedence Reordering (AWP) and Row-Hit Aware Write Precedence Reordering (RAWP). Both techniques can achieve more throughput improvement than the baseline, and RAWP can achieve similar read row buffer hit rate to the baseline at the same time.

Third, I leverage the information provided by Differential Write and extend memory scheduling with a fine-grained power budgeting technique for better utilization of power budgets [105]. Due to PCM's high write power and write current, a practical PCM memory is typically designed with a limit on the number of concurrent writes [15, 40] (termed "power budget" in this thesis). The increased parallelism achieved by non-blocking bank design and my memory scheduling enhancements may result in higher total write power and exceed the original power budget of the baseline bank design. Simply increasing the power budget to accommodate this increased parallelism is expensive, because supplying high write current and voltage is challenging to the charge pumps of the write drivers, and the transient high current causes noise on the power line [15, 40]. On the other hand, keeping the original power budget as in the baseline bank design may require limiting the number of concurrent PCM write requests, which contradicts the throughput improvement techniques I have proposed. To preserve the benefits I gain from memory scheduling enhancements without increasing the power budget, I propose Bit Level Power Budgeting (BPB) technique. BPB leverages the information from Differential Write to get a better estimation of power demands and to choose write configurations flexibly. As a result, BPB shows more throughput improvement under the same power budgets, showing its better utilization of power budgets.

Fourth, I propose techniques to improve QoS tuning ability on high-priority applications [102], which are conceptually at higher levels than the memory scheduling enhancements and power budgeting techniques I propose. When multiple applications are running concurrently, their memory requests can interfere with each other and cause longer read latencies. It is often desirable to be able to control this "memory slowdown" for high-priority applications. However, PCM's long write latency worsens the interference and degrades this tuning ability. A high-priority application can still suffer from significant read latency increases, even if its requests are given highest priority. Hence the tunable range of high-priority application's read latency increase is limited, indicating poor QoS tuning ability. I propose two techniques, Request Preemption and Row Buffer Partitioning, to mitigate this issue. Experiments show that my techniques can extend the tunable range of high-priority application's read latency increase by 1.7× ∼10×, indicating better QoS tuning ability.

1.2 CONTRIBUTION

In summary, the contributions of this thesis are as follows:

- I propose Differential Write, a circuit-level technique to remove unnecessary (redundant) bit changes in PCM writes. Simple and effective, Differential Write can remove 85% of total PCM bit writes and is beneficial for both endurance and energy. With Differential Write and two simple wear-leveling techniques, I demonstrate that PCM-based main memory is practical in terms of lifetime. This is one of the first attempts to use PCM in main memory and study its feasibility. Moreover, Differential Write opens new opportunities to upper level (memory scheduling) techniques.
- I evaluate the energy-delay savings of PCM-based main memory through modeling, and describe the considerations on peripheral logic selection. This is among the first work that provides energy/delay modeling of PCM memory as well as its architectural evaluation.
- I identify the challenges to PCM memory that are caused by its disadvantageous write operations. Based on my circuit-level technique Differential Write, I propose a series of memory scheduling techniques for PCM memory to address these challenges. These techniques include

memory scheduling enhancements, fine-grained power budgeting and QoS tuning ability improvement schemes. They form a comprehensive effort towards successful application of PCM memory.

- I evaluate the effectiveness of my proposed techniques with detailed simulations. I describe my experimental infrastructure which could be helpful to further research in this area.

1.3 THESIS ORGANIZATION

The rest of this thesis is organized as follows: Chapter 2 presents background information. Chapter 3 discusses the related work. Proposed techniques are presented in Chapter 4 through Chapter 7. Chapter 8 presents my experimental infrastructure. Chapter 9 describes future research directions and concludes the thesis.

2.0 BACKGROUND

2.1 PHASE CHANGE MEMORY (PCM)

Phase Change Memory, or PCM, is one type of non-volatile memory that exploits the unique behavior of phase change material to store information. Although the technology emerged recently, the theory of phase change material has its origins early in 1960s when Ovshinsky reported a reversible change in resistivity upon a change in phase in certain glasses [71]. The first Phase Change Memory device was announced in the September 28th, 1970 issue of Electronics by Neale et al [64,66]. In the following years, the advance of semiconductor manufacturing technology enabled the development and application of PCM. For example, phase change material is already widely used in rewritable CDs and DVDs, in which the same alloy is used as the PCM memory developed by Numonyx [66]. By exploiting the electrical resistivity of phase change material, PCM is drawing increasing interest recently, as it can be used as a memory cell and organized into memory array similar to DRAM.

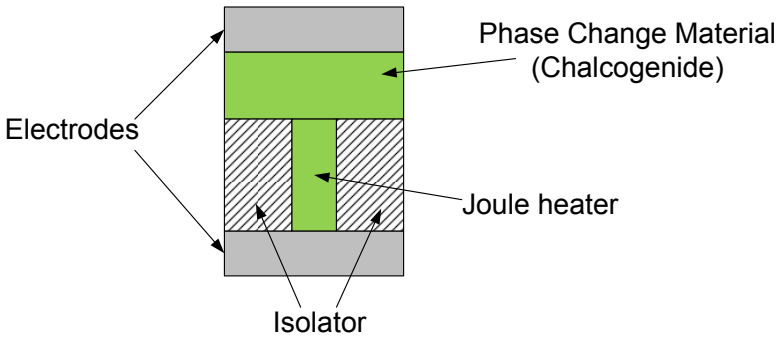


Figure 3: Structure of PCM cell.

A conventional DRAM cell uses a capacitor to store a bit of information. Analogously, a PCM cell uses phase change material to remember a bit. The phase change material is one type of

chalcogenide alloy, such as $Ge_2Sb_2Te_5$ (or GST in short), which has two stable physical states: amorphous and crystalline. In the amorphous state, the material is highly disordered and exhibits high resistivity. In the crystalline state, the material has a regular crystalline structure and exhibits low resistivity. PCM exploits the difference in resistivity between these two states of the material to store data. Typically, a cell in the amorphous state (high resistance) is regarded as a logic "0" (a.k.a. RESET state), and a cell in crystalline state (low resistance) is regarded as a logic "1" (a.k.a. SET state). Unlike DRAM that relies on constant refresh to retain its data, the state of GST is preserved even after the cell is powered off, meaning that PCM is non-volatile. PCM also has good data retention time. In a prototype by Bedeschi et al., a ten-year data retention was reported [3].

Figure 3 illustrates the structure of a typical PCM cell. A layer of chalcogenide (GST) is sandwiched between two electrodes. A joule heater is placed between GST and the bottom electrode. The structure forms a PCM cell, which appears as a resistance in the circuit.

Reading data from a PCM cell involves sensing the resistance level of the cell. This is done by applying a small voltage across the two electrodes so that the resistance of GST can be measured. This process is non-destructive and has negligible heat stress compared to write operations.

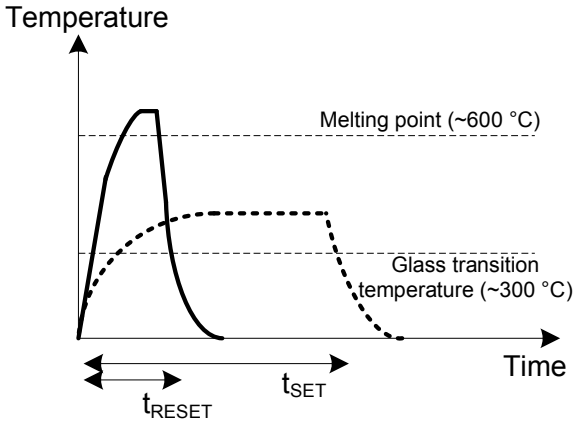


Figure 4: Write operation of PCM cell.

Writing a PCM cell, on the other hand, requires changing the physical state of its GST material. This is done by injecting current into the junction of the GST and the heater to induce phase change through joule heating. When heated above its crystallization temperature (~ 300°C) but below its melting temperature (~ 600°C) over a period of time, GST turns into a low-resistance crystalline state (which corresponds to logic "1" or SET state). When heated above its melting point and

quenched quickly, GST turns into a high-resistance state (which corresponds to logic "0" or RESET state). Figure 4 illustrates the two process of PCM write operations. Though writing a PCM cell incurs high operating temperature, the thermal cross-talk between adjacent cells at 65nm is shown to be negligible even without thermal insulation material [74]. Similar to the multi-level flash memory, the phase change material can also be programmed into four or more distinct states, forming a multi-level PCM cell that can represent four or more values [39, 74].

Despite the differences with DRAM cells, PCM can still use a similar array structure as DRAM arrays. It can use the same peripheral logic such as decoders, row buffers, request/reply networks etc. as the DRAM array [50, 60]. Figure 5 illustrates a typical structure of a 2×2 PCM cell array. Each PCM cell is connected between an access transistor and a bitline (BL). The access transistor is controlled by the wordline (WL). In order to access a PCM cell, its wordline is selected to enable the access transistor, which forms a path between the cell's bitline and ground. Read or write operations on the PCM cell are then done by applying different voltage pulses on the bitline. Also, as can be seen from Figure 5, selecting a wordline enables the access transistor of a series ("row") of PCM cells. By controlling the bitlines using a column mux, read or write operations can be performed on selected cells in that row. For example, in Figure 5, if the bottom WL is selected, the access transistors of the bottom two cells are enabled. And if the bitline on the right side is selected by a column mux (i.e., the other bitline is left floating), read or write operations are performed on the bottom-right cell (circled out in the figure).

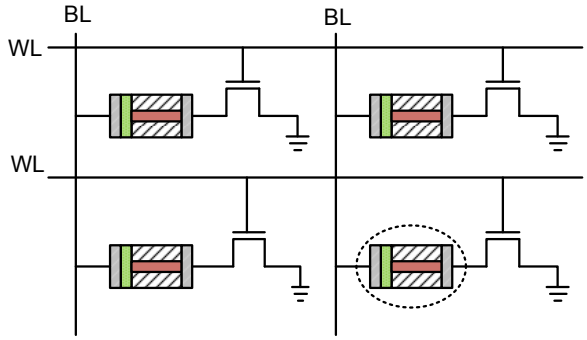


Figure 5: PCM cell array [40, 50, 60].

There are two main options for the access of the device in a PCM cell: a transistor or a diode. A diode has a simpler structure, and hence it is good for cell density. However, a diode cannot satisfy the high write current requirement beyond sub-100nm technology [14]. Also the scaling rule of a diode is not so clear as NMOS [74]. Lastly, the diode-based cell has been reported to be more vulnerable to errors induced by writing data to adjacent cells because of bipolar turn-on of the nearest-neighbor cells [68]. Taking all of the above, especially the scalability, into consideration, I assume transistor-based PCM cells in my experiments. My assumption was also confirmed by Li et al. in a later work [54].

PCM's advantages. Like DRAM, PCM memory is byte-addressable, giving it an big advantage over current NAND Flash technology that only supports block accesses. PCM offers comparable read latency (∼50ns) as DRAM [67]. Typical area size of PCM cell is $5{\sim}8F^2$ [47], meaning that PCM has similar density to DRAM.

With shrinking feature size, DRAM is facing serious scaling problems as it is bounded by the limitation in cell-bitline capacitance ratio. DRAM has been found to be difficult to scale below 40nm [61]. PCM, on the other hand, offers much better scalability: When a PCM cell shrinks, the volume of the GST material shrinks as well, resulting in less write current [66]. Hence PCM provides a truly scalable solution compared to conventional DRAM. A recent prototype by Liang et al. demonstrated the viability of the PCM cell reaching 2.5nm technology node [55].

Like NAND Flash, PCM is non-volatile. In theory, a PCM cell only consumes energy when it is accessed (read or write). This makes it possible to build memory chips with low leakage, which is crucial to meet the low-power requirements of future memory systems. Moreover, PCM uses physical states instead of electrical charge to represent data, making it much less vulnerable to the soft errors caused by alpha particles or cosmic radiation [66].

PCM's disadvantages. Due to repeated heat stress applied to the phase change material, PCM has limited number of write cycles (i.e., write endurance). A single cell can typically sustain $10^8 \sim 10^9$ [25,39,99] writes before a failure can occur. While this is much better than NAND Flash, it could be a big concern if PCM is used in main memory. As we will see in Chapter 4, a PCM memory without any lifetime improvement technique may last only ∼100 days running a typical SPEC CPU program.

PCM's write operation is also slower than its read operation due to the heating/cooling process during a write. A typical latency of writing a PCM cell is 90ns∼150ns [48,103]. What makes things worse is that PCM write incurs high current injection (e.g., 0.6∼1mA [50]), resulting in high write

power (2.88∼4.8mW per bit [50]). When PCM is used as memory, a multi-bit memory write (e.g., writing a 512-bit memory line) is usually completed in several *rounds*, with each round writing part of the line. This makes PCM's per-access write latency even longer. For example, Numonyx reported a 1us (1000ns) page write latency for its PCM prototype [67]. This long per-access write latency and high write power have created some major issues when using PCM in main memory, as I have discussed in Chapter 1.

Another issue that has recently drawn attention is the resistance drift in multi-level cells (MLC) [11, 29]. After a multi-level PCM cell is programmed, its resistance may increase and saturate over time, which is termed "resistance drift" [101]. Previous studies have indicated that the phenomena is caused by the structural relaxation of chalcogenide material, which is a thermally-activated, atomic rearrangement of the amorphous structure [11]. Resistance drift can cause a PCM cell to change to another state at runtime, causing a soft error. A study by Awasthi et al. has shown that the drift may occur as soon as 1.81 seconds after a cell is programmed [2]. In response to this problem, architectural techniques have been proposed to mitigate the issue without high overhead, such as [101] and [2]. In this thesis, I assume a single-level cell PCM which does not have the resistance drift issue. However, the techniques proposed in this thesis are independent of PCM cell technology. They can be extended to multi-level cell (MLC) and are orthogonal to the drift-mitigating schemes.

2.2 MEMORY SCHEDULING

2.2.1 Memory Scheduling Basics

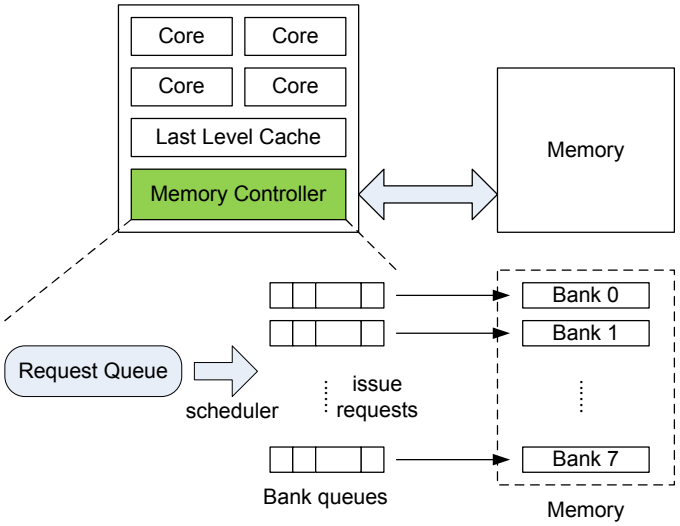


Figure 6: Overview of memory controller and memory scheduling.

A memory controller is a key unit that determines how memory requests from the last level cache are dispatched to individual memory banks and get served. In a typical design, memory requests arriving at memory controller are first buffered in a request queue (as shown in Figure 6). To achieve high throughput, main memory can typically serve multiple outstanding requests at a time. Hence it can be viewed as having multiple logic banks that can work concurrently. The memory controller dispatches the requests from the request queue to the logic banks according to certain *memory scheduling policies*. To hold the dispatched requests, each logic bank has a corresponding small bank queue in the memory controller. When a request is dispatched by the memory controller, it is moved from the request queue to the corresponding bank queue. Requests in a bank queue are then issued and served in order.

When a logic bank is actively serving a request, the request at the head of its bank queue cannot be issued until the logic bank becomes idle (ready). Due to the shared command bus, the memory controller can issue up to one request in each memory cycle (for DDR2-800, a memory cycle is 2.5ns). Hence if more than one logic bank are idle (ready), their head-of-bank-queue requests are issued in a round-robin fashion.

2.2.2 Parallelism-Aware Batch Scheduling

In this section, I introduce the parallelism-aware batching scheme (PAR-BS) [63] developed by Mutlu et al. as it is used as the baseline scheduler in my memory scheduling-related techniques. PAR-BS aims at promoting inter-thread and inter-bank parallelism. The scheduling scheme includes two parts: Request Batching and Parallelism-Aware Within Batch Scheduling.

Request Batching. PAR-BS organizes memory requests into *batches* and ensures all requests in the current batch are served before the next batch is formed. When forming a new batch, PAR-BS marks up to `Marking-Cap` requests per (logical) bank per thread. This batch forming procedure ensures fairness and promotes both inter-thread and inter-bank parallelism (as the number of requests from each bank or thread tends to be balanced).

Parallelism-Aware Within Batch Scheduling. Within each batch, PAR-BS strives to promote both row buffer hits and parallelism. It employs a ranking system and put requests with higher rank first. Request ranking is determined as follows (in the order of significance):

1. **Batched** – Requests that are in the current batch (i.e., marked requests) are prioritized over requests that are not.
2. **Row-hit** – Requests that will get a row hit are prioritized over requests that will get row miss.
3. **Thread-rank** – Requests from threads with higher ranks (which will be explained later) are prioritized over requests from lower-ranked threads.
4. **FCFS** – Finally, older requests are prioritized over younger requests.

PAR-BS uses a *Max-Total* rule to determine **thread ranks**. For each thread, PAR-BS finds the maximum number of marked requests to any given bank, called "max-bank-load". Thread ranks are determined with the following rules:

1. Max rule: A thread with a lower max-bank-load is ranked higher than a thread with a higher max-bank-load.
2. Tie-breaker total rule: For each thread, the scheduler keeps track of the total number of marked requests, called "total-load". If threads are ranked the same under Max rule, the thread with lower total-load is ranked higher than a thread with higher total-load.
3. Finally, all remaining ties are broken randomly.

By combining the above techniques, PAR-BS achieves the goals of parallelism, starvation-free and row buffer hit promotion. The batching mechanism in PAR-BS ensures fairness and avoids

starvation. The `Marking-Cap` limitation of its batch forming procedure helps promote inter-bank and inter-thread parallelism. The within batch scheduling (request ranking) promotes row buffer hits.

PAR-BS is used as the baseline scheduler in my PCM memory designs. My memory scheduling enhancements are developed on top of PAR-BS to improve throughput (Chapter 5) and QoS tuning ability (Chapter 7). My power budgeting technique (Chapter 6) is also evaluated with an enhanced PAR-BS scheduler.

3.0 RELATED WORK

3.1 PROTOTYPING, CHARACTERIZATION AND MODELING OF PCM

Much device-level work related to PCM has been conducted, including device characterization, prototyping and modeling.

Device prototyping. Various prototypes have been fabricated to verify and demonstrate the feasibility of PCM. In [14], Cho et al. presented a 64Mb 1T1R PRAM using 0.18-μm technology. A 64Mb PRAM with confined cell structure was presented in [13], which shows good potential for high density. In [99], Yeung et al. adopted the $Ge_2Sb_2Te_5$ confined structure to their 64Mb PRAM to achieve low reset currents. Large sensing margin and reasonable endurance (10^9 cycles) were demonstrated by their prototype. Chen et al. proposed an ultra-thin phase-change bridge (PCB) memory cell which simplifies scaling and offers potential for both fast write and good data retention [10]. In [69], a cell current regulator scheme and multiple step-down pulse generators were employed to improve write performance of PRAM. And in [68], a 512Mb PRAM with $5.8F^2$ cell size, fabricated using 90nm technology was demonstrated. In [50], Lee et al. presented a 512Mb diode-switch PRAM with 266MB/s read throughput. Bedeschi et al. presented a 256Mb multi-level cell (MLC) test-chip using 90nm PCM technology [3]. In [39], Kang et al. reported two key factors of two-bit (four-level) cell operation in PCM, namely write-and-verify and moderate-quenched writing.

Annunziata et al. presented a 90nm embedded PCM (ePCM) technology in [1]. In this work, a 4Mb ePCM is integrated in CMOS platform with few additional masks and minimum process tuning. This confirms the technology as a viable solution for replacing conventional floating gate non-volatile memory (Flash) in embedded systems. In follow-up work in [18], Sandre et al. presented a prototype with a cell size of $0.29\mu m^2$, 1.2V 12ns read access time and 1MB/s write throughput.

A 58nm 1.8V 1Gb PCM prototype with 6.4MB/s programing bandwidth was presented by Chung et al. in [15]. The prototype was implemented in a 58mn PRAM process with a low power double-data-rate non-volatile memory (LPDDR2-N) interface. Notably, this prototype has built-in Flip-N-Write [12] technique (which was called data comparison write with inversion flag, or DCWI in their work).

In [16], a fully-integrated 512Mb PCM chip using 90nm CMOS technology is presented. The prototype demonstrates a multi-level cell PCM, in which each cell represents 2 bits.

Li et al. proposed a reconfigurable sensing scheme with the flexibility to change reading precision of analog resistance levels for a multi-level PCM cell [53]. A 2M-cell chip was fabricated in 90nm CMOS technology as a proof-of-concept. This prototype employs a single dynamic reference voltage and fixed WL voltage, eliminating the complicated multiple reference circuits. The sensing scheme allows for the changing of sensing precision, and can be applied as a built-in self-test (BIST) unit.

Liang et al. presented a cross-point PCM cell working close to its scaling limit with the ultra low reset current of 1.4uA [55]. The prototype utilized a carbon nanotube as the bottom electrode, and demonstrated potential viability of PCM for highly scaled ultra-dense memory at the 2.5nm node.

Wen et al. presented a non-volatile lookup-up table (LUT) using PCM cells [96]. The LUT, fabricated in IBM 90nm CMOS technology, can perform programmable and non-volatile logic functions with 1V supply. A 453.4ps average propagation delay (i.e., read latency of the LUT) was measured in their test.

Characterization and modeling. In [47], Lai et al. reviewed the device structure and characteristics of phase-change memory. Pirovano et al. studied the scalability of PCM and indicated that reset current scales down with the device size [74]. Thermal cross-talk between adjacent bits was also investigated in this work [74]. Mohammad et al. studied PCM failure modes and defective behaviors, and developed fault models for PCM [60].

Braga et al. studied the stability of multi-level PCM cell's intermediate states programmed by partial-SET [7]. Their study observed significant dependence of retention properties upon the pulse duration and amplitude of a partial-SET pulse.

E. Bozorg-Grayeli et al. studied the thermal properties of the electrode materials used in PCM [6]. In many designs, significant heat loss occurs through the electrodes. This work proposed a multilayer electrode stack and investigated its thermal properties. The multilayer electrode stack

offers greater thermal resistance than single-material electrodes due to the presence of multiple thermal boundary resistances, reducing heat loss from the device and may potentially lower the programming current [6].

Li et al. studied the relative advantages of different driving (access) devices for PCM [54]. According to the study, a diode is more advantageous for larger technology nodes, but the MOSFET shows potential superiority in many aspects when scaling to the 65nm technology node and beyond.

Papandreou proposed a novel iterative write-and-verify programming scheme that uses both partial-SET and partial-RESET pulses [72]. Their scheme allows PCM's resistance to be changed in both directions, and can achieve favorable trade-offs between latency and robustness.

Faraclas et al. studied the SET and RESET operations of PCM using 2-D finite-element simulations with rotational symmetry [22]. Their simulation results predict voltage/current levels which are comparable to experimental data.

Prior art and my work. Previous device-level work has provided me with important insights into the mechanism of PCM operations (e.g., PCM write operation), as well as key device parameters that are used in my experiments (e.g., latency, power), either directly or indirectly. A recent prototype presented by Chung et al. [15] incorporated the Flip-N-Write [12] technique. I adopted the technique in my power budgeting scheme to further improve the utilization of power budgets. While my Differential Write idea was developed independently, it was also confirmed by a Samsung prototype in [49].

3.2 ARCHITECTURAL INNOVATIONS ON PCM MEMORY

System architecture. Wu et al. explored the design space of hybrid cache architecture (HCA) using different memory technologies in [98]. Embedded DRAM (EDRAM), Magnetic RAM (MRAM) and PCM (PRAM) were explored in both 2D and 3D cache architectures. Two types of hybrid cache architectures (HCA) were evaluated:

- inter cache level HCA (LHCA), in which the levels of cache hierarchy can be made of different memory technologies.
- intra cache level or cache region-based HCA (RHCA), where a single level of cache can be partitioned into multiple regions, each using different memory technology.

An LRU-based cache line migration scheme was proposed for RHCA, and drowsy mode was proposed to be used in the slow region of a cache level to save power. Their experiments showed that LHCA, RHCA and 3D-RHCA achieved IPC improvements of 7%, 12% and 18% respectively.

In [80], Qureshi et al. proposed a PCM/DRAM hybrid memory system. The system used PCM along with DRAM buffer as main memory, which is similar to my architecture. Four techniques were proposed in this architecture:

- Lazy-Write: On a page fault, a page loaded from hard disk is only written to the DRAM buffer to avoid slow PCM write. The page is written to PCM only when it is evicted from the DRAM buffer.
- Line-Level Write Back (LLWB): Instead of writing PCM in page granularity, LLWB only writes back dirty lines within a page. This requires adding a dirty bit to each line of a page.
- Fine-Grained Wear-Leveling (FGWL): Lines in each page are stored in the PCM in a rotated manner. A pointer called WearLevelShift is generated randomly to decide how the lines are shifted (e.g., WearLevelShift=1 means Line 0 is stored at physical address of Line 1 and so on).
- Page Level Bypass (PLB): In case of streaming applications, the OS can enable PLB to avoid storing pages in PCM to preserve PCM lifetime.

Lee et al. proposed area-neutral architectural enhancements to make PCM competitive with DRAM [48]. This work proposed an important idea of row buffer organization in PCM memory: use a narrow row buffer entry to mitigate per-access PCM write energy, and use multiple row buffer entries to improve locality and write coalescing. In this design, the row buffer of PCM memory is organized like a small cache with multiple entries instead of a single long entry. Experiments showed that this narrow, multi-entry row buffer organization is beneficial for both latency and energy. This idea is also adopted in my architecture. In addition, this work proposed a partial write scheme to improve lifetime of PCM memory. Partial write marks dirty words of each memory write and only writes the dirty words of the request. Since a dirty word may still contain many redundant bit changes, partial write cannot fully exploit the opportunity of value locality.

In [75], a Morphable Memory System (MMS) was proposed by Qureshi et al. The scheme was based on observation that memory requirements varies between workloads, and systems are typically over-provisioned in terms of memory capacity. During a phase of low memory usage, MMS allows some of the multi-level cell bits to be used as single-level cells (which have lower latency). And when the workload requires high memory capacity, these cells can be restored to multi-level

cells to obtain high density. Their experiments showed that 95% of all memory requests were served in low latency mode by MMS, resulting in 40% better performance.

Performance. In [36], Joshi et al. proposed a scheme called Mercury to improve write speed and energy in PCM multi-level cells. Mercury used a state-aware adaptive programming scheme when writing a multi-level PCM cell. The idea is similar to the bi-directional programming in [72]. Programming a cell from SET to RESET (S2R) was achieved by a SET pulse followed by multiple short RESET pulses (which gradually increase the cell resistance). Programming from RESET to SET (R2S) was achieved by a short RESET pulse followed by multiple short pulses with step down amplitude (which gradually decreases cell resistance). The S2R method has latency advantage if target resistance is low, while R2S method has the advantage of programming reliability (better distribution of resistances). Mercury selects R2S and S2R based on the target resistance level to achieve fast and energy efficient MLC writes.

To mitigate PCM's long write latency, Qureshi et al. proposed write-cancelation and write-pausing schemes [76]. In these schemes, an on-going write request on a bank can be canceled or paused, giving way to a subsequent read request to improve read latencies. The write request is restarted (in case of write-cancelation) or resumed (in case of write-pausing) afterwards. Write-cancelation can be implemented with both single-level cells and multi-level cells. Write-pausing, on the other hand, only applies to multi-level cells because it assumes an iterative write-and-verify process which is used in multi-level cell programming. In write-pausing, an on-going PCM write can be paused between its two iterations (meaning that the cell is left in an intermediate state). The paused write is then resumed from the intermediate state it was left off. In both techniques, only one request is served in each bank at a time. Writes and reads are still in serial but not in parallel. Hence, they do not help to improve the throughput of the PCM memory.

Lifetime. In [77], Qureshi et al. proposed Start-Gap wear-leveling for hybrid memory system. In Start-Gap, memory is augmented with an extra line (GapLine) which contains no useful data and two registers (*Start* and *Gap*). All lines can be regarded as forming a circular buffer. *Start* indicates the location of first logical line in the memory, and *Gap* always points to GapLine. GapLine is moved by 1 location periodically, which is accomplished simply by copying the contents at location [*Gap*-1] to GapLine and decrementing *Gap*. Everytime when *Gap* reaches *Start*, it means all lines in memory are shifted by 1 location and hence *Start* is incremented by 1. The procedure is illustrated in Figure 7.

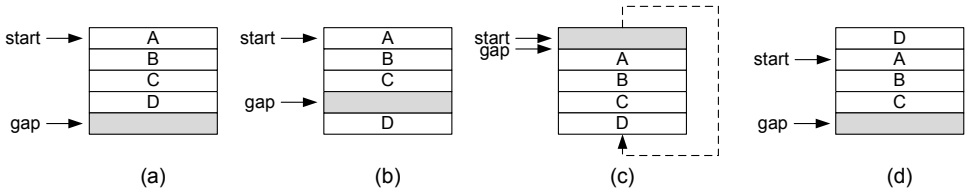


Figure 7: Start-Gap wear-leveling in a memory containing 4 lines [77].

Start-Gap has the advantage of low storage and computation overhead (as logical-physical address mapping can be calculated using simple arithmetic operations). However, it has the shortcoming of slow line movements. If a memory contains a large number of lines, it takes a long time for every line to be shifted. This means that Start-Gap is only effective in small memory regions. To overcome this shortcoming, the author partitioned the large memory into smaller regions, each running Start-Gap independently. This forms the Region-Based Start-Gap (RGSG). The scheme can be further augmented with address space randomization, in which addresses are randomly shuffled before reaching PCM memory. Randomized Start-Gap was reported to achieve 97% of the theoretical maximum lifetime. Experiments also showed that RBSG can make PCM memory robust to malicious attacks.

Seong et al. proposed Security Refresh [90], which is more robust against malicious attack than Start-Gap. Security Refresh exploits the property of XOR function to perform address remapping. A Memory Address (MA) is XORed with current key to generate the Remapped Memory Address (RMA). When the key is changed, memory address mappings are swapped in pairs. For example, if the key is changed from 00 to 01, address 00 will be remapped to 01 and address 01 will be remapped to 00. Hence, Security Refresh can support randomized address mapping without high computation or storage overhead.

Qureshi et al. also presented an improvement to his Start-Gap scheme in [79] called Adaptive Wear-Leveling. Adaptive Wear-Leveling used an online attack detector to detect malicious behaviors. An attack was indicated by the Intra Write Distance, which is the number of writes between two consecutive writes to a same line. In normal applications, this number was found to be much larger than 1000. In order to detect an Intra Write Distance less than 1000, a small LRU stack was used to approximate the 1K-entry write access history. The gap movement frequency was

made adaptive to application behavior: for normal applications it is less frequent, and for malicious applications it is more frequent to make it more difficult for an attacker to wear out the memory.

In [24], Ferreira et al. presented a set of techniques to improve the lifetime of PCM memory. Their architecture also assumed a DRAM buffer (which is organized as a page cache) before PCM memory. Several techniques were proposed in this work to improve PCM lifetime:

- N-Chance page replacement policy: When evicting a page from the DRAM buffer, try to find a clean page among the N least recently used pages. If such a page does not exist, then LRU page is evicted.
- Page partitioning and Read-Write-Read (RWR): A page is partitioned into sub-pages, each having its own dirty bit. When a page is written back to PCM, only the dirty sub-pages are written. Within sub-pages, RWR is applied to further avoid unnecessary bit changes. RWR performs a read before write to compare the old value with new value, and only writes the bits that are actually changed. A read is then performed after the write for fault detection.
- Page swapping: Pages are swapped on DRAM page cache writebacks. A global write counter is used to determine the swap condition (whether a swap is needed), and the target page for swapping is selected randomly. The advantage of swapping pages on a writeback is that it only introduces one additional page write.

In [19], Dong et al. proposed Wear Rate Leveling which takes endurance variation into consideration. Instead of using write counts only, they use "wear rate" (write counts over endurance, or W_i/E_i) to accommodate the different endurance between strong and weak cells in wear-leveling. A naive solution to generate the address mapping is to sort the write counts and endurance rates. However, this may generate lots of swaps during a remap. To reduce the number of swaps, the author formulated the problem into a Maximum Weight Perfect Matching Problem in a bipartite graph, which can be solved using the Hungarian Algorithm. As a result, their scheme achieved the same optimal wear rate as a naive implementation with 68% less swapping.

Bock et al. studies an interesting phenomena called useless write-backs [4]. A useless write-back occurs when a dirty cache line that belongs to a dead memory region (a memory region that is no longer used by the program) is evicted. Since the evicted data is not used by the program again, such write-backs can be safely avoided to improve endurance and energy consumption. The author

proposed algorithms for counting useless write-backs in different memory regions: heap, global data and stack. Experiments showed that avoiding useless write-backs can save up to 19.8% of energy consumption and improve endurance by up to 26.2%.

In [23], Ferreira et al. presented modeling techniques to analyze the trade-offs for endurance management based on the anticipated distribution of cell lifetimes. Two general endurance strategies are considered: physical capacity degradation (PCD) and physical sparing (PS). In PCD, all memory is used and, as cells wear out, usable memory size is reduced. In PS, damaged cells are replaced with operational spare cells from excessive capacity. The authors studied the two strategies under four different distribution of cell lifetime: constant, linear, normal and bimodal. The models presented in this work can be used to determine how much redundancy is needed when a sparing endurance strategy is adopted.

Kong et al. proposed to use counter-mode encryption in PCM for privacy protection [46]. However, encryption can significantly reduce the effectiveness of partial writes [48] and Differential Writes [103]. To mitigate the problem, the authors added several block-level counters beneath each line-level counter and tracked dirty blocks within the line. Upon a write-back, only the block-level counters of dirty blocks are incremented (meaning that only dirty blocks are re-encrypted). The line-level counter is incremented when any of its block-level counter overflows (causing the entire line to be re-encrypted). This simple extension made partial write to be effective again, which is beneficial for endurance. Another scheme proposed in this work is the adaptive ECC management. Instead of applying ECC with fixed strength, the authors proposed to track memory wear-out using encryption counters and gradually increase ECC strength as memory wears out. Experiments showed that adaptive ECC can extend the lifetime of PCM with low storage overhead and performance penalty.

Jiang et al. proposed LLS, a cooperative integration of wear-leveling and salvaging technique for PCM memory [33]. Salvaging schemes are used to remap failed PCM lines with spare lines to extend the lifetime of PCM memory. Current wear-leveling and salvaging schemes have not been designed to work cooperatively. Salvaging causes non-contiguous PCM space, which complicates wear-leveling and incurs large overhead. To solve this issue, the authors proposed Line-Level mapping and Salvaging design that can provide a contiguous PCM address to OS and applications. Experiments showed that LLS achieves 24% longer lifetime with negligible overhead.

In [34], Jiang et al. studied the over-RESET problem in PCM memory. PCM cell endurance was found to be heavily dependent on the RESET current. To accommodate process variation,

larger-than-optimal RESET current is typically used, resulting shortened lifetime. In this work, the authors proposed to leave a small number of hard-to-reset cells unused (dormant) and use ECC to rescue these cells. This results in smaller RESET current and longer cell lifetime. When weak cells start to wear out, these dormant cells are used as replacement (which requires voltage upscaling due to their larger RESET current). Experiments showed that their scheme can reduce PCM write power by 33% and extend lifetime by up to 102%.

Power. In [12], Cho et al. proposed Flip-N-Write technique to limit PCM write power. When writing a line, Flip-N-Write (FnW) writes either the original value or its inversion, whichever results in fewer bit flips. Therefore, Flip-N-Write guarantees that no more than half of the bits in each write are changed. This means that under the same power budget (number of concurrent bit writes), Flip-N-Write can finish a write faster because it doubles the write width. For example, if the current power budget allows 64 concurrent bit writes, a 512-bit write request will take 8 rounds to finish in baseline (each round writing 64 bits). Flip-N-Write, on the other hand, can finish the same write in 4 rounds because it can write 128 bits per round (in which up to 64 bits are actually changed). This results in better performance under same power budget.

Park et al. proposed a power management scheme for DRAM/PCM hybrid memory [73]. The scheme mainly focused on reducing refresh power of the DRAM buffer. Each line in DRAM buffer was associated with a time-out counter to control its eviction. An evicted line no longer received refresh to reduce refresh power. The authors proposed a runtime-adaptive time out control scheme to minimize total energy consumption of DRAM and PCM. In addition, DRAM is bypassed for the first read access to filter out accesses with low spatial locality. And dirty data in DRAM are assigned with larger time out values in order to keep them longer in DRAM to reduce PCM writes.

In recent work by Hay et al., a power budgeting technique called "power token" was proposed [26]. The technique aims at PCM's high write power, and ensures that the number of concurrent bit writes do not exceed a power budget (which is defined by the DDR interface they used). The authors tried to count the number of bit changes in each write request for estimation of power demands. In order to avoid transmitting information between memory DIMM and the memory controller, their scheme made two approximations: 1) They use the data in last-level cache to track bit changes, which means it is only accurate on the first write back to the last-level cache. In case of multiple write backs to last-level cache, the number of bit changes are accumulated,

forming a conservative estimation. 2) They reduce the granularity of bit change counters to 3 bits to reduce overhead, which also reduces accuracy. Essentially, the power token scheme is a power gating technique enhanced with conservative bit change estimations.

Reliability. Unlike DRAM, PCM is less susceptible to transient faults (soft errors). Instead, hard errors (due to endurance) are more important in PCM. Schechter et al. proposed a new approach to error correction optimized for PCM called Error Correct Pointer (ECP) [85]. ECP exploits the nature of hard errors (permanent and immediately detectable at write time). It corrects errors by permanently encoding the locations of failed cells into a table and assigning cells to replace them. ECP was reported to provide longer lifetimes than previously proposed solutions with equivalent overhead.

In [86], a hardware-efficient multi-bit stuck-at fault error recovery scheme called SAFER was proposed. SAFER exploits the key attribute that a failed cell with a stuck-at value is still readable, making it possible to continue to use the failed cell to store data and reduce hardware overhead for error recovery. SAFER dynamically partitions a data block and ensures that there is at most one fail bit per partition. It then uses single error correction techniques per partition for error recovery. Comparing to ECP, SAFER was reported to increase the number of recoverable fails and achieves better lifetime improvement with smaller hardware overhead.

Yoon et al. proposed another improved scheme called FREE-p to handle both hard error and soft error [100]. In contrast to coarse-grained approaches, FREE-p used fine-grained remapping (FR) for failed blocks. Based on a key observation that even a deemed dead block still has many functional bits that can store useful information, FREE-p embeds a remapping pointer in it. Hence the mapped-out block itself (which is otherwise useless) is used as free storage for remapping information. And to mitigate the penalty of accessing remapped blocks, a remapped pointer cache was proposed. FREE-p was reported to improve lifetime over ECP by up to 26%, with less than 2% performance overhead.

Zhang et al. proposed a resistance drift resilient architecture for multi-level cell PCM called Helmet [101]. The scheme was motivated by observation that resistance drift has a strong correlation with data patterns. Through adaptive data inversion and rotation, Helmet tried to store the majority of values in their drift-insensitive formats (e.g., 00 or 11) rather than drift-sensitive formats (e.g., 01 or 10). Experiments showed that Helmet can reduce error rates by 87%.

Platform evaluation tools. Caulfield et al. presented an architecture of prototype PCIe-attached storage array built from emulated PCM storage called Moneta [8]. Moneta was imple-

mented using PCIe-attached array of FPGAs and DRAM. The authors used FPGA to implement scheduler and a set of configurable memory controllers, allowing them to emulate non-volatile memory (such as PCM) accesses. As the prototype used interface PCIe, it was targeted for storage applications instead of as main memory.

In a later work from Jung et al., a Persistent RAM Storage Monitor called PRISM was proposed [37]. PRISM provided a useful tool for studying Persistent RAM (PRAM) storage behavior, and could be used to guide the design of PRAM storage device (PSD). When a user application generates read or write system calls, PRISM traces the calls and generates statistics of low-level access activities occurring at the storage subsystem. PRISM consists of two parts: the frontend tracer and the backend PSD simulator (PSDSim). The frontend tracer captures I/O system calls and passes them to the PSDSim. PSDSim then simulates the behavior of PSD based on its configuration (e.g., PRAM technology, storage capacity, etc.), and generates the statistics which can be used for further analysis. PRSIM was demonstrated to be a versatile and useful tool for studying interactions between application/OS and PRAM device accesses, and is extensible through user-defined tracers.

Prior art and my work. My proposed technique, Differential Write, was one of the early attempts of applying PCM in main memory [103]. Using Differential Write along with simple wear-leveling techniques, I demonstrated that PCM-based main memory is feasible in terms of lifetime. Many other wear-leveling techniques were proposed later on, each with different advantages such as lower overhead or better resilience against attack.

My memory scheduling techniques are developed on a PCM memory with a similar architecture to [80] and [24]. I also adopted the multi-entry row buffer design proposed in [48] and developed techniques on top of it. My power budgeting technique incorporates both Differential Write and Flip-N-Write [12] to achieve better utilization of power budgets. The memory scheduling enhancements proposed in my thesis aim at improving throughput and QoS tuning ability of PCM memory. Although they use some ideas that are similar to write-cancelation and write-pausing [76], they are very different schemes that address different problems. Write-cancelation and write-pausing only improve read latency, while my techniques address the throughput and QoS tuning ability problems.

The power token technique [26] proposed by Hay et al. aims at a similar issue to my BPB technique: PCM's write power. However, my BPB technique is significantly different than power token in the following aspects:

- Although power token is enhanced with estimations of bit changes in write requests, it still only serves as a power gating technique. It only ensures that PCM write power does not exceed a pre-defined limit. On the contrary, BPB has the ability of choosing write configurations for write requests and skipping redundant rounds in addition to the simple power gating function.
- BPB makes more accurate estimation of bit changes as it collects and processes information locally. On the contrary, power token make estimations by conservatively accumulating the number of bit changes in last-level cache. BPB also has finer granularity in its estimations (bit level) than the 3-bit approximation used by power token.

My techniques are also orthogonal and additive to other architecture schemes for PCM memory, as they target different problems. For example, my memory scheduling enhancements can be applied on top of a PCM memory using reliability improvement techniques.

3.3 MEMORY SCHEDULING POLICIES

Memory scheduling policy is a crucial part of the memory controller to determine how memory requests are dispatched. Different memory scheduling policies could have different design goals, e.g., throughput or fairness. And to achieve their design goals, memory scheduling policies may need to reorder their requests. This implies that data dependencies among memory requests have been resolved before they arrive at memory controller.

FCFS and FR-FCFS. The simplest memory scheduling policy is first-come-first-serve (FCFS), meaning that requests are dispatched in the order as they arrive at the memory controller. Old requests are always dispatched earlier than young requests. Apparently, this simple policy cannot achieve good throughput. A simple enhancement is called first-ready FCFS (FR-FCFS) [83, 84]. Instead of dispatching requests in order, FR-FCFS prioritizes requests that will get row buffer hits to improve bandwidth utilization and memory system performance [83,84]. The problem with FR-FCFS is starvation: it is possible that an application with high locality is always served first because its memory requests are row hits, causing starvation of applications with low locality. Hence many of the recent memory scheduling polices are designed with fairness or starvation avoidance in mind.

NFQ. Nesbit et al. proposed a network fair queuing (NFQ) scheme to achieve QoS and fairness in scheduling memory requests from multiple applications [65]. In NFQ, each thread i is allocated with a fraction ϕ_i of memory system's bandwidth. This is accomplished by using virtual clocks for

each thread and making each virtual clock run ϕ_i times slower than the memory frequency. When scheduling requests, NFQ always prioritizes requests with earliest virtual finish-time, which is the virtual time when a memory request will finish in its thread's virtual clock. However, NFQ has a potential starvation problem when there are threads with bursty behavior [62, 81]. As discussed in by Rafique et al. in [81], NFQ delays the calculation of the finish-time of memory requests to just before they are about to be issued. This could cause finish-time tags to be out-of-order, meaning that a younger request might receive a smaller finish-time than an older request. In the worst case, younger requests can keep getting issued, causing starvation or long latencies for some of the older requests.

SFQ. To solve the possible starvation problem in NFQ, Rafique *et al.* proposed the start-time fair queuing instead of the virtual finish time fair queuing [81]. Instead of calculating exact finish times, SFQ abstracts memory transactions into units (which greatly reduces complexity) and uses start-time when scheduling outstanding requests. SFQ assumes each memory request takes one unit of service time. Each thread is assigned with a service interval which is the reciprocal of its weight. SFQ always selects the request from the thread with earliest scheduled service time (start-time). Experiments showed that SFQ achieves performance improvements of 21% over NFQ.

STFM. Mutlu and Moscibroda proposed the stall-time fair memory (STFM) scheduling policy in [62]. The goal of STFM is to equalize the DRAM-related slowdown experienced by each thread due to interference with other threads. STFM uses stall-time to compute fairness metrics instead of finish-time to improve fairness and throughput over the NFQ scheme [62]. The memory slowdown in STFM is defined as T_{shared}/T_{alone}. T_{shared} is the memory-related stall-time experienced by a thread when it is running along with other threads. T_{alone} is the estimated memory-related stall-time the thread would have if it had run alone. STFM tracks T_{shared} for each thread, and estimates T_{alone} as $T_{shared} - T_{interference}$. The $T_{interference}$ is the extra stall-time the thread experiences due to service of requests from other threads. STFM then computes unfairness index as the ratio between maximum slowdown and minimum slowdown. At runtime, STFM sets an unfairness cap α on unfairness index. When unfairness is below α, STFM uses FR-FCFS for high performance. And when unfairness exceeds α, STFM turns to a fairness-oriented scheduling policy, picking the requests from maximum slowdown first. STFM was reported to reduce unfairness on memory slowdown from 5.26× to 1.4×, with an average system throughput improvement of 7.6%.

PAR-BS. In a more recent work, Mutlu et al. proposed parallelism-aware batching scheme (PAR-BS) [63]. PAR-BS organizes incoming requests into batches, and schedules the requests

within a batch to exploit inter-thread and inter-bank parallelism. The scheme consists of two parts: Request Batching and Within-Batch Scheduling. Request Batching marks `Marking-Cap` of requests per thread per bank to form a new batch. And Within-Batch Scheduling uses request ranking policy to schedule requests within a batch. By combining the two techniques, PAR-BS achieves the goals of parallelism, starvation-free and row buffer hit promotion. PAR-BS is used as baseline scheduler in my memory scheduling techniques, hence I discuss more of its details in Background chapter (Section 2.2.2).

TCM. In a later work in [45], Kim et al. proposed Thread Cluster Memory scheduling (TCM). The TCM scheme categorizes threads into two "clusters", memory intensive and non-intensive, according to their miss-per-kilo-instructions (MPKI). It then divides bandwidth between two clusters, and employ different policies in them. In non-intensive cluster, TCM prioritizes low MPKI threads to improve system throughput; while in memory-intensive cluster, TCM uses a rank shuffling mechanism to ensure fairness. According their experiments, TCM further improved performance and fairness over PAR-BS by 7.6% and 4.6% respectively.

Read-While-Write. The concept of parallelizing a read with a long write was first implemented as a read-while-write (RWW) operation in NOR flash [94], where Flash memory on a chip is divided into code and data areas. A RWW NOR flash device is essentially two or more flash memories on a chip. One memory is used to store data and the other is for code. When the data memory is written, code can be read out from the other memory. The concept is also adopted in recent PCM prototypes [95], where a PCM chip is divided into smaller partitions, and two partitions can be simultaneously active, one being read and the other being written.

Prior art and my work. Existing memory scheduling policies are designed for DRAM-based memory systems, which do not consider the properties of PCM. My memory scheduling enhancements proposed in Chapter 5 extends existing memory scheduler by exploiting intra-bank parallelism on non-blocking PCM bank. Since my memory scheduling enhancements are designed as extensions to existing memory scheduler, they are orthogonal and additive to other existing memory scheduling schemes. For example, my memory scheduling enhancements can be applied on top of TCM [45] to get more throughput improvement. My Bit Level Power Budgeting technique (Chapter 6) can also work with other existing memory schedulers without complex firmware changes due to its decoupled design.

Comparing to the non-blocking bank design used by my memory scheduling enhancements, the RWW concept allows one write in each bank and shows much less throughput improvement in my

experiments, as PCM's throughput is mainly limited by its write requests. My experiments also show that memory scheduling enhancements are necessary to enjoy the benefit of more parallelism provided by non-blocking bank design.

3.4 ASYMMETRIC READ/WRITE ACCESSES

PCM's asymmetric read/write speed is analogous to a property of Flash, in which write operations are much more expensive than read operations. There have been techniques proposed for Flash to deal with this asymmetric read/write access property.

In [28], Hu et al. proposed a scheme to reduce Flash write operations on embedded system. They assumed an embedded CMP with scratch-pad memories (SPM) and a Flash-based main memory. Two techniques were proposed to reduce Flash write operations. 1) Data migration: when evicting dirty data from a scratch-pad memory, they try to move the data to another scratch-pad memory with free space instead of writing back to Flash-based main memory. 2) Data recomputation: if data to be written back to main memory will be read back by another task later, they discard this write-back operation and recompute the data when it is used again. In other words, they are trying to reduce write operations in the expense of more computation at runtime. Their experiments show that the number of writes was reduced by 59% on average. However, their scheme was based on an embedded system with a special configuration (using scratch-pad memories and Flash-based main memory). So it may not be suitable for the large scale, byte-addressable memory on general purpose computer.

On et al. proposed Lazy-Update B+ tree for Flash drives [70]. B+ tree is a commonly used index-structure to expedite query processing in database systems. The purpose of their scheme is to optimize the update (write) operations of B+ tree. In Lazy-Update, update requests to B+ tree are deferred and buffered in an update pool, and committed later in groups. Their experiments show that Lazy-Update can reduce the number of page writes by half while preserving the query efficiency.

Flash drives usually have an internal write buffer. Improving the management of write buffer can help mitigate the expensive Flash write operations. In [35], Jin et al. proposed a write-aware buffer management scheme. The scheme uses a state transition diagram to identify write-intensive pages, and tries to retain the write-intensive pages in the write buffer. Their experiments showed

that number of writes was reduced by up to 30%. Shi et al. proposed a cooperating scheme [87] in which virtual memory and write buffer are managed in a cooperating way: When virtual memory manager tries to write back a dirty page, it prefers to write back the pages that are in the write buffer. The cooperating management scheme reduced the number of erase and write operations by 44.7% and 28.6% respectively.

Another research area on mitigating Flash write costs is the coding scheme. In a multi-level cell Flash memory, each cell can store multi-bit data. When writing a multi-level Flash cell, its value can only be changed from a lower state to a higher state (e.g., 0→1, 1→2). Resetting a cell back to 0 requires an erase operation, which is very expensive. This property of multi-level Flash memory can be formulated into a Write-Asymmetric Memory (WAM) model [32]. For a WAM with multiple variables (cells), coding schemes were proposed to improve the number of times the variables can be written or rewritten in order to avoid the high cost of erase operations. For example, Jiang et al. proposed floating codes in [32]. And in [5], a buffer coding scheme was proposed by Bohossian et al. These coding schemes were developed for multi-level cell Flash with irreversible transit properties, and may not be suitable for PCM memory.

Prior art and my work. Since Flash memories are usually used as storage devices, many of the schemes for mitigating the asymmetric read/write costs in Flash are at different levels than my memory scheduling techniques. Hence they are mostly orthogonal to my techniques. For example, the data migration/recomputation scheme and the Lazy-Update scheme are at the system software level, and can be used along with my memory scheduling enhancements. The buffer management schemes for Flash memory can inspire useful ideas to improve the management of the DRAM buffer above PCM memory. However, improving throughput of PCM memory is still necessary because 1) A memory with lower throughput would require a much larger cache/buffer in order to achieve equivalent system performance (as I will discuss in Section 6.7); 2) Caching may not be effective when the working set of the workload is large, or when running streaming workloads.

4.0 LIFETIME IMPROVEMENT OF PCM MEMORY

In this chapter, I propose a circuit-level technique called Differential Write to remove unnecessary (redundant) bit changes in PCM writes (Figure 8). Combining Differential Write and two simple wear-leveling techniques, I demonstrate that PCM-based main memory is practical in terms of lifetime [103]. Differential Write is not only beneficial for endurance and energy, but also opens new opportunities for upper level memory scheduling. Architectural modeling and evaluation of PCM memory are also conducted. My work is among the first attempts to use PCM in main memory, and forms the base of other components in my research.

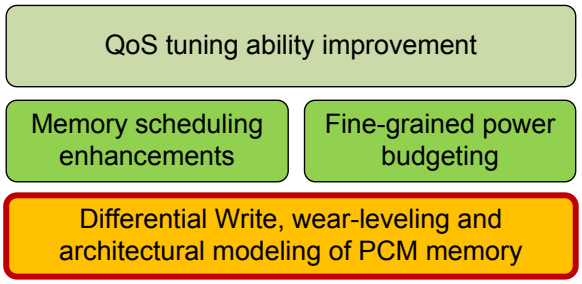


Figure 8: Overview of my research – Differential Write.

4.1 LIFETIME PROBLEM OF PCM MEMORY

PCM's limited write endurance raises the concerns of its lifetime when used in main memory. To illustrate the problem, I test the "unprotected" lifetimes of a PCM main memory using a variety of benchmarks including SPEC2K, SPEC2006, and SPECWeb. The memory lifetime running each benchmark is estimated assuming the PCM main memory is constantly accessed at a rate generated

by this benchmark. Figure 9 shows the projected lifetime of PCM memory when no lifetime enhancement techniques are employed. In reality, such rate may vary with different workloads running in the system. The number of rewrite cycles for a PCM cell is assumed to be 10^8. As shown in Figure 9, the results range from 25 days for `mcf` to 777 days for `specweb-banking`, and the average is only 171 days. Hence in order to make PCM memory practical, lifetime improvement techniques are needed to extend PCM lifetime to an acceptable level.

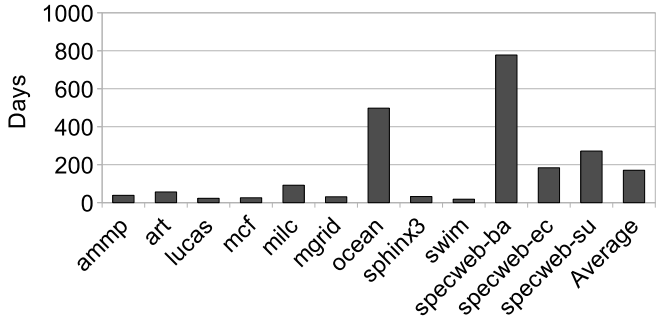


Figure 9: Lifetime of PCM memory when used without any lifetime improvement techniques.

4.2 REDUNDANT BIT WRITES

To improve the lifetime of PCM memory, my first step is to reduce the total number of bit writes. In a conventional memory write, every bit in the request is written once. However, I observed that a great portion of these bit writes are redundant. That is, in most cases, a write into a cell did not change its value. I term this "redundant bit writes". These bit writes are hence unnecessary, and removing them can greatly reduce the write frequency of the corresponding cells. Fig. 10 shows the percentages of redundant bit writes for different benchmarks. They are calculated as the number of redundant bit writes over the total number of bits in write accesses. The 'SLC' series represents redundant bit-writes in a single level PCM cell, i.e., each cell stores either '0' or '1'. The 'MLC-2' and 'MLC-4' series represent 2-bit and 4-bit multi-level PCM cells. That is, each cell stores 4 (MLC-2) or 16 (MLC-4) binary values. The number of rewrite cycles for both SLC and MLC's are assumed to be 10^8.

From the results, it it clear that all benchmarks exhibit high percentages of redundant bit writes. For single-level cells, the statistical bit-write redundancy is 50% if writing a '0' and '1'

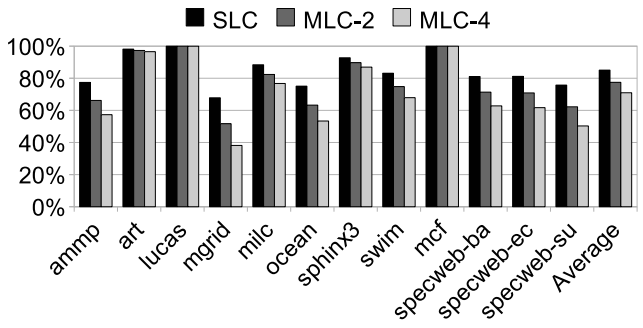


Figure 10: Percentage of redundant bit writes for single-level and multi-level cells.

is equally likely. For MLC-2 and MLC-4 cells, the redundancy probabilities are 25% and 6.25% ($\frac{1}{2}^4$) respectively. However, the measured redundancies for real workloads are much higher than the theoretic values, showing interesting *value locality*. The redundancy ranges for SLC, MLC-2, and MLC-4 cells are 68∼99%, 52∼99%, and 38∼99%, with an average of 85%, 77% and 71% respectively. Avoiding these unnecessary bit writes can reduce the total number of bit writes to PCM memory, which is beneficial for its lifetime. This inspired me with the idea of removing redundant bit writes, which leads to the Differential Write technique.

4.3 DIFFERENTIAL WRITE

Differential Write removes redundant bit writes by performing read and compare before each write access. In PCM operations, reads are much faster than writes, so the delay increase here is less than doubling the latency of a write. Also, write operations are typically less critical than read operations, so increasing write latency has less negative impact on the performance of the workload.

The comparison logic can be simply implemented by adding an XNOR gate on the write path of a cell, as illustrated in Figure 11. The XNOR output is connected to a transistor which can block the write current when the write data equals the currently stored data. Delay and power overhead of the XNOR gate were measured to be 75ps and 199μW, and were counted in my evaluations.

After applying the Differential Write technique, the lifetime of PCM memory is extended to 770/592/510 days, or 2.1/1.6/1.4 years, for SLC/MLC-2/MLC-4 respectively on average, as shown

in Figure 12. Moreover, Differential Write results in localized bit changes inside each row. This provides more room for wear-leveling and new opportunities for upper level memory scheduling (which will be discussed in following Chapters).

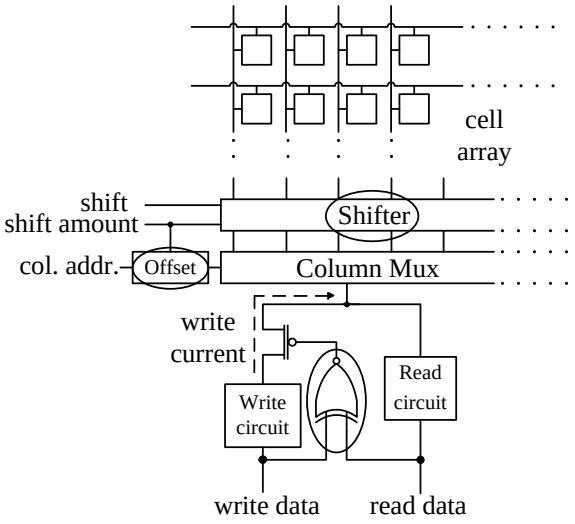


Figure 11: Implementation of Differential Write and row shifting.

4.4 WEAR LEVELING

Even though redundant bit-write removal achieved up to 5 times lifetime extension, the resulting 1.4∼2.1 years of lifetime is still too short for main memory. The reason is that the memory updates happen too locally: the bulk of writes are destined to only a small number bits, creating an extremely unbalanced write distribution. Therefore, those "hot" cells fail much sooner than the rest of the cells. For this reason, the next step I take is wear leveling.

4.4.1 Row Shifting

Each PCM memory write contains a line, or a "row" of bits (512 bits in my experiments). After applying Differential Write, the bits that are written most in a row tend to be localized, rather than spread out. Hence, I apply a simple shift mechanism to even out the writes in a row to all cells instead of a few cells. Simulation results indicate that it is not beneficial to shift on bit granularity,

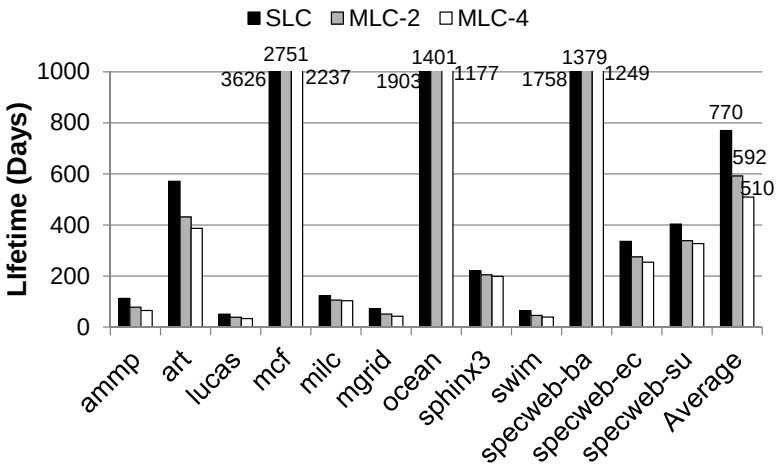


Figure 12: Lifetime (days) after applying Differential Write.

because hot cells tend to cluster together. For example, least significant bits are written more often than most significant bits. Shifting on too coarse a granularity is also not helpful since cold cells might be left out. For those reasons, I found that shifting by one byte at a time is most effective.

In addition, my simulation results also indicate that it is not advantageous to shift on every write because once a line is shifted, writing it back may incur more bit changes than without the shift. Hence, I perform the shift periodically to amortize its cost. Moreover, a workload does not have equal accesses to every memory pages. Some pages are "hotter" than others, and some pages are accessed more balanced for every line than others. The best shift interval varies significantly from page to page. Hence, I select two representative memory pages from each of the four categories: hot, medium hot, medium balanced and unbalanced pages for a workload. They are selected based on the write counts of the pages, and the standard deviations of writes among all lines in a page. I do not consider cold pages as their lifetimes are much longer and tuning the shift interval should not be disturbed by statistics from them.

I varied the row shift interval from 0 (no shift) to 256 (shift one byte on every 256 writes), and collected the resulting lifetimes averaged from all selected sample pages. We found that the results from each individual benchmark varies greatly. For example, the `specweb-banking` favors an interval of 16 writes while this interval generated the lowest lifetime for `mcf`. Hence, I summarized

the lifetime for different shift intervals averaging over all benchmarks. I used not only arithmetic mean, but also geometric mean and harmonic mean to give more weight to low-lifetime workloads. The results are plotted in Figure 13.

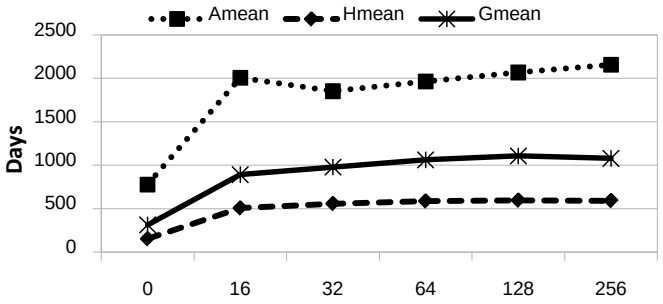


Figure 13: Lifetime with different row shift interval averaged over all benchmarks.

As can be seen from the figure, the best shift interval is 256 writes, as it generates the highest lifetime for all means. However, there is no need to increase the interval as 1) both geometric and harmonic means have leveled off, and 2) longer interval incurs more counter bits and hardware overhead corresponding to each line. With such a wear leveling for each row, the lifetime of PCM memory increased to 5.9/4.5/3.9 years for SLC/MLC-2/MLC-4 respectively (as shown in Figure 14), which is still not sufficient for commodity systems.

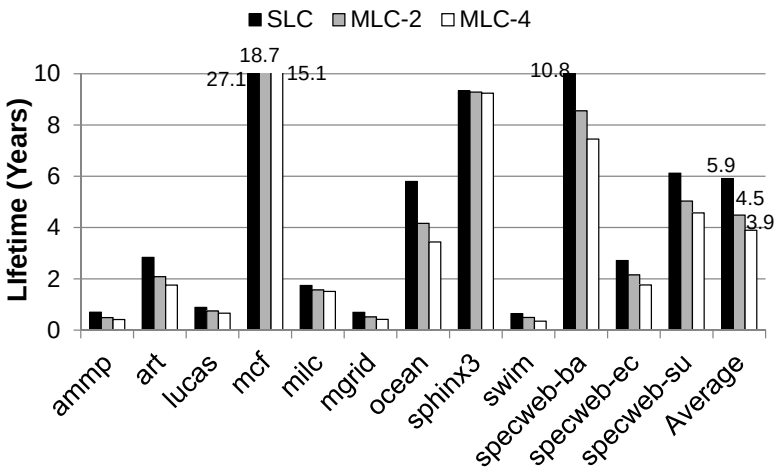


Figure 14: Lifetime (years) after applying Differential Write and Row Shifting.

Row shifting can be implemented with an additional row shifter, along with a 6-bit shift offset per 64B line, as shown in Figure 11. The shifter, together with the column mux and the shift

offset, performs the complete shift of a line. On a read access, the data is reset to its original position before being sent out. On a write access, new data is first shifted in the pre-write read for comparison. Delay and power introduced by the shifter in each memory access were measured to be 400ps and 795μW respectively. Both these overheads are considered in my simulation results.

4.4.2 Segment Swapping

The next step of wear leveling I took is at a coarser granularity. Row shifting rotates the bytes inside each memory line. However, write distribution in some workloads can be highly unbalanced, meaning that some hot pages are written more often than others. For example, Figure 15 shows the distribution of memory writes for benchmark `mcf` in 10 seconds of simulated time. The X-axis is the memory space in unit of 1MB segment, and the y-axis denotes the number of writes in logarithmic scale in each segment. Row shifting has only limited effect on hot pages that have significant amount of writes than others. Those pages will still fail sooner even though each line in it has balanced writes. Therefore, I develop a coarse granularity wear leveling mechanism that periodically swaps memory segments of high and low write accesses.

Figure 15: Memory write distribution for mcf in 10 seconds of simulated time.

The main parameters I need to determine for segment swapping are segment size and swap interval. To select a proper segment size, I experimented with small sizes such as one, or several pages. The main difficulty is that the metadata to keep track of the page writes would be too big. For example, for a 4GB memory with 4KB page size, 1M of page write counters need to be maintained. This is not only a big storage overhead but also it requires long latency at runtime for sorting the counters to find cold pages for swapping. Therefore, I enlarged the segment sizes and

experimented with a number of options. For each segment size, I varied the the swap intervals in number of writes, and collected the resulting lifetime improvement factors as depicted in Figure 16.

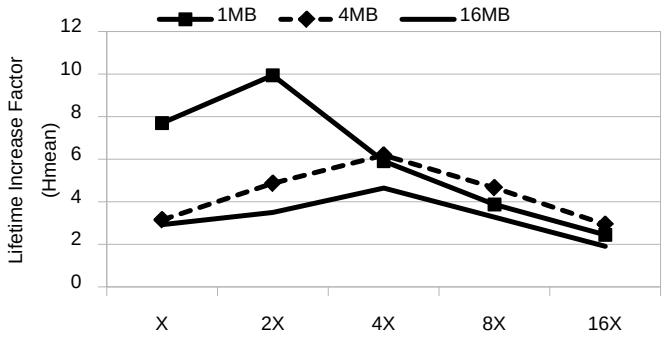


Figure 16: Effect of segment size and swap interval on lifetime (Hmean).

The X-axis shows different swap intervals in unit of a base interval ('X'). This is because larger segments should use larger intervals so different segment sizes have different base intervals. The Y-axis shows the lifetime improvement factors averaged over all benchmarks under test. The goal of wear-leveling is to distribute memory writes more evenly in the memory space. Its effectiveness can be measured by the maximum number of writes to any memory segment (W_{max}), lower is better. The lifetime improvement factor is computed as the ratio of W_{max} values between with and without wear-leveling. I used lifetime improvement factor in the results because 1) it is a direct result of wear-leveling; 2) estimating absolute lifetime for wear-leveling scheme requires tracking number of writes at per-bit granularity in the entire memory space, which is too expensive to experiment.

The results in Figure 16 clearly show that larger segment sizes do not benefit from wear leveling. This is because swapping larger segments introduces higher overhead in terms of extra writes. For example, the overhead for 1MB, 4MB, and 16MB segments on their base swap intervals are 2.8%, 5.6% and 5.2% respectively. Therefore, the best option is the 1MB segment size with 2X swap interval which corresponds to 2×10^6 writes. Note that segment swaps bring overhead mostly in performance. Endurance wise, each cell involved in the swapping is written only for *one more time*, which has negligible impact on the lifetime.

Segment swapping should be implemented in the memory controller. A mapping table between the "virtual" segment number generated by the core and the "true" segment number for the actually location of the requested segment is maintained. The size of this table depends on the capacity of

the main memory. For a 4GB main memory, the 1MB segment size results 4K entries in the table, and each entry stores $A - 20$ bits where A is the width of the physical memory address.

For each segment, two control data are tracked: 1) *write_count* which counts the number of writes to the segment; and 2) *last_swapped* which remembers when this segment was swapped last time. A cold segment may be picked multiple times for swapping in a short period of time. Keeping *last_swapped* can prevent the cold segment from being selected again too soon. In addition, the segment swap should not happen too frequently within a short amount of time. If there are many swap requests, the controller can delay them because the time a swap should happen is not very critical. Such a design can even out segment swaps so that they do not have significant impact on system performance. I set a global constant *swap_throttle* as the threshold of the number of swaps that can happen in a fixed period of time. A *swap_queue* is used to record the awaiting swap requests if they cannot be serviced promptly.

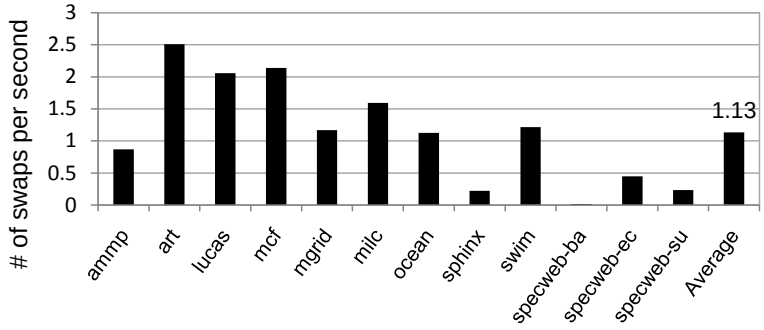


Figure 17: Frequency of segment swapping.

For the swap configuration I selected — 1MB segment swapped on every $2{\times}10^6$ writes, I found the average number of swaps occurred per second is 1.13, as shown in Figure 17. When a swap occurs, the memory has to stall for 2×1MB/64B PCM writes, assuming each PCM write is 64B. As I will show in Section 4.6, each PCM write takes up to 156.55ns (36.28ns for pre-write read + 120.27ns for write). Therefore, the memory is unavailable for 1.13 × 156.55ns × 2× 1MB/64B = 5.76ms per second. This amounts to 0.576% performance degradation to the running workloads in the worst case. This overhead can be decreased by using two row buffers in the memory controller to read and write the exchanging lines in parallel, reducing the overhead by half to 0.288%.

4.5 LIFETIME IMPROVEMENTS

After applying all three techniques I proposed (Differential Write, Row Shifting and Segment Swapping), the lifetime of a PCM main memory can be extended greatly. I assumed 4GB PCM main memory in the test and 10^8 of cell write endurance. As shown in Figure 18, average lifetime of PCM memory is extended to 13∼22 years. Detailed results are listed in Table 4.5. These results indicate that with my Differential Write and wear-leveling techniques, PCM memory is practical in terms of lifetime.

Note that all three techniques are necessary to extend the lifetime of PCM memory because their effects are multiplied. As I have already shown in Figure 12 and Figure 14, applying Differential Write and Row Shifting can extend lifetime to 5.9 years. If Segment Swapping is applied alone (without Differential Write or Row Shifting), it can only extend lifetime to 1.7 years by harmonic mean (the 2nd column of Table 4.5). These results prove that each technique is an essential component of my scheme.

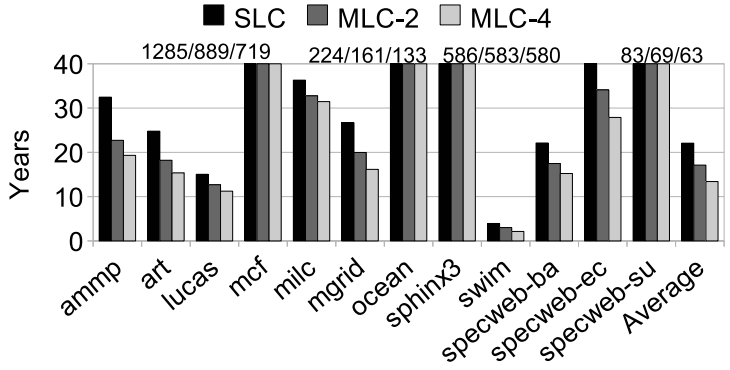


Figure 18: PCM memory lifetime after applying all three techniques.

4.6 ARCHITECTURAL MODELING AND EVALUATION

As one of the early attempts of using PCM in main memory, I developed an energy and delay model for architectural evaluation of PCM memory.

PCM has been implemented using the same architecture as DRAMs [40, 50, 69]. They share similar peripheral circuits but differ in the implementation of cells. Hence, the methodology I

Table 2: Lifetime (years) after applying Differential Write, Row Shifting and Segment Swapping.

Benchmarks	SLC(segment swap only)	SLC	MLC-2	MLC-4
ammp	4.9	32.5	22.7	19.3
art	1.3	24.8	18.2	15.4
lucas	1.1	15.0	12.7	11.2
mgrid	1.8	26.7	20.0	16.2
milc	2.3	36.3	32.8	31.5
ocean	52.8	224.4	161.2	133.1
sphinx3	5.5	586.2	583.3	580.9
swim	0.3	3.9	3.0	2.1
mcf	3.3	1285.4	889.0	719.5
specweb-banking	4.4	22.1	17.5	15.2
specweb-ecommerce	8.0	42.9	34.1	27.9
specweb-support	10.2	83.9	69.1	62.7
Amean	8.0	198.7	155.3	136.3
Hmean	1.7	**22.1**	**17.1**	**13.4**

used is to simulate the essential circuits such as the cell, bitlines, wordlines, read/write circuits etc. in HSPICE, and then replace the CACTI results related to those essential circuits with our HSPICE results. In other words, I only use the skeleton of the CACTI DRAM model, and fill in the contents with HSPICE PCM model. [1] This method also provides a fair comparison because PCM and DRAM will use the same floor plan. Feature size used in my model is 45nm, and memory capacity is 4GB. The numbers produced by these simulations were then applied in my architectural model for evaluation of PCM memory.

4.6.1 Peripheral Device Type

CACTI-D provides three types of devices defined by ITRS: high performance (HP), low standby power (LSTP), and low operating power (LOP). These devices are used in the peripheral and global support circuitry such as input/output network, decoders, sense amplifiers etc. The HP devices have the highest operating speed. They have short gate length, thin gate oxide, and low V_{th}. The downside is that HP devices have high dynamic and leakage power. The LSTP devices on the other hand are designed for low leakage power. They have longer gate length, thicker gate oxides, and higher V_{th}. Naturally, these devices are slower than for HP devices. The LOP devices are between HP and LSTP devices in terms of performance. The advantage is that LOP devices provide lowest dynamic power among the three.

DRAM typically uses Low Standby Power (LSTP) transistors for the peripheral circuitry to lower its leakage energy. However, it may not be a good choice for PCM because LSTP yields worst performance among the three choices. More importantly, PCM does not require LSTP devices to lower its leakage because it is non-volatile. When the memory is idle, much of the peripheral circuitry can be powered down without losing the stored data. This is the most distinct benefit of using PCM with performance advantageous peripheral circuitry, which is feasible for DRAM.

Between HP and LOP, HP transistors are faster but consume more energy, in both dynamic and leakage energy. LOP transistors are slower but consume less total energy. My experiments showed that HP devices have too high leakage at runtime to be acceptable even for PCM. Hence LOP devices were chosen for the peripheral circuits in my model.

[1] Circuit design and CACTI/HSPICE simulation were done by Bo Zhao (boz7@pitt.edu) at ECE department, University of Pittsburgh.

4.6.2 Energy/Delay Model

Table 3 lists the latency and energy parameters used in my architectural model for PCM memory evaluation. The numbers were derived from CACTI and HSPICE simulation. In the table, PCM's burst read is a little faster than DRAM. This is mainly because of two reasons. First, the peripheral logic of PCM (LOP devices) are faster than that of the DRAM (LSTP devices). I have discussed the trade-offs of these choices earlier. Second, the DRAM reads are destructive. DRAM needs to restore (write back) the data to cells before the next access to the same bank. This is a process to regenerate full-rail values on the high capacitive bitlines, which takes some time. These operations increase the DRAM memory's random access time. In contrast, PCM-based memory can easily handle burst reads by nature because PCM reads are not destructive.

Table 3: Latency and energy parameters used in my architectural model.

	Latency (ns)		Energy (nJ)	
	PCM	DRAM	PCM	DRAM
Read	36.28 (row miss)	20.04 (row miss)	10.68 (row miss)	12.17 (row miss)
	6.47 (row hit)	9.33 (row hit)	3.77 (row hit)	12.06 (row hit)
Write	90.27 (0)	20.04	0.0268 (0)	14.48 per row (64B)
	120.27 (1)		0.013733 (1)	

Write latency. With Differential Write, PCM's write latency is not fixed. If a write request is completely redundant (i.e., every bit of the line to be written is same as the old data in memory), then this PCM write request can be terminated after the pre-write read and comparison operations, resulting in shorter latency. This is the major difference in PCM's latency model.

Write energy. With Differential Write, the per access write energy of PCM is also not fixed. It is calculated as follows:

$$E_{pcmwrite} = E_{fixed} + E_{read} + E_{bitchange}$$

E_{fixed} is the "fixed" portion of energy charged for each PCM write including row selecting, decoding, XNOR gates, etc. This part is 4.1nJ per access as measured from HSPICE simulation. E_{read} is the energy to read out the wordline for comparison. This part is approximately 1.075nJ which includes the energy spent in the row shifter as shown in Figure 11.

The $E_{bitchange}$ part depends on the proportion of updated bits ($0 \rightarrow 1$ or $1 \rightarrow 0$): $E_{bitchange} = E_{1\rightarrow0}N_{1\rightarrow0} + E_{0\rightarrow1}N_{0\rightarrow1}$. As listed in Table 3, $E_{1\rightarrow0}$ and $E_{0\rightarrow1}$ are 0.0268nJ and 0.013733nJ respectively. Therefore, per access write energy for PCM (nJ) can be expressed as:

$$E_{pcmwrite} = 5.175 + 0.0268 \times N_{1\rightarrow0} + 0.013733 \times N_{0\rightarrow1}$$

What can be seen from here is that Differential Write can significantly reduce PCM's write energy, in addition to its benefits of endurance. For example, from the statistics in Figure 10, 85% of bit writes are removed (for SLC). If we assume writing a '0' or '1' is equally likely, the total write energy per "row" is 6.73nJ including the energy on the circuit illustrated in Figure 11. This is significantly lower than the per access write energy of DRAM.

4.6.3 Experimental Setup

I evaluated PCM-based main memory with a 4-core CMP (with parameters listed in Table 4) in Simics [56] simulation environment. I set the core frequency as 1GHz because I assumed a 3D architecture (discussed below) which is subject to tight thermal constraints. Trace-driven simulation was used for lifetime analysis and execution-driven simulation was used for studying energy and performance. For the latter, I used the GEMS [57] simulator with both Ruby (detailed cache and memory simulator) and Opal (out-of-order core simulator) activated. I enhanced the memory module to model both the latency and energy consumption of PCM and DRAM. Due to the limitation of Simics g-cache module (which is discussed in Section 8.2.1), my traces include memory accesses generated by user applications and operating system, but they do not include memory accesses that are generated by DMA operations.

I assumed that the memory is stacked onto the 4-core chip, similar to the PicoServer architecture [41]. I chose a 3D architecture to evaluate PCM because if it was used as an off-chip memory, the latency on the CPU-memory bus would diminish the latency problem of PCM. The memory is organized as one memory module with 4 ranks, assuming each rank is one layer in a 3D stack. The 3D architecture used in my experiments does not include a DRAM buffer. Hence my lifetime results can be regarded as the "worst-case" scenario in which PCM is used as main memory directly. I implemented burst read mode in the row buffer. The parameter for PCM and DRAM are taken from the HSPICE simulations and rounded to integer numbers of memory cycles.

Table 4: Hardware parameters of a 4-core CMP.

Processor core	4-OOO-core, each core runs at 1GHz
L1 Cache	Private L1 cache (32K I-cache and 32K D-cache), 64-byte lines, 4-way set associative, 3 cycles access time
L2 Cache	Shared L2 cache, 4MB, 64-byte lines, 16-way set associative, 6 cycles access time
Memory controller	one controller, next to core 0
Memory size	4GB memory
Memory organization	1 DIMM, 4 Ranks/DIMM, 16 Banks/Rank; use top 16 bits as row number; each rank is a layer of 3D stacking of on-chip memory
Interconnect Network	2×2 mesh network

4.6.4 Evaluation Results

4.6.4.1 Energy savings Figure 19 shows the dynamic energy of DRAM (left bar) and PCM (right bar). They are further broken down to initial reads, burst reads, writes and refreshes. PCM's dynamic energy is only 47% of the DRAM's dynamic energy, achieving a 53% reduction. The greatest reductions come from 1) the write energy because of Differential Write; and 2) the burst read energy due to PCM's non-destructive read. Moreover, PCM-based memory does not require refresh logic and the associated energy costs because of its non-volatile nature.

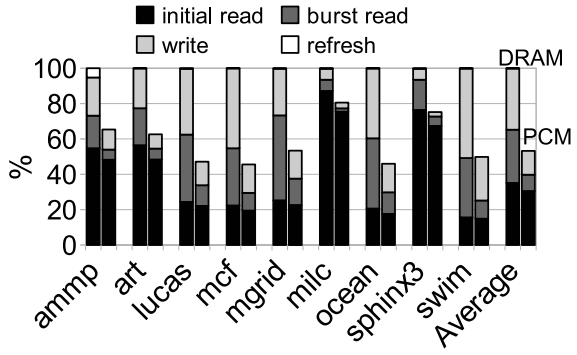


Figure 19: Breakdown of dynamic energy savings.

The leakage savings come from two sources: 1) cell leakage reduction due to the non-volatility of PCM cells; and 2) the power gating of peripheral circuits when the memory is idle. During this time I power down the peripheral circuits because I will not lose contents in the memory. This can save significant energy in the peripheral circuits because PCM uses LOP devices which have higher leakage when active than the LSTP devices for the DRAM peripheral circuits. Note that even for memory intensive workloads such as *mcf*, the leakage saving is nearly 40%. The combined effect results in large leakage energy reductions, as illustrated in Figure 20. On average, PCM-based main memory achieved 74% savings, when compared with a DRAM-based memory that has already been optimized for low leakage.

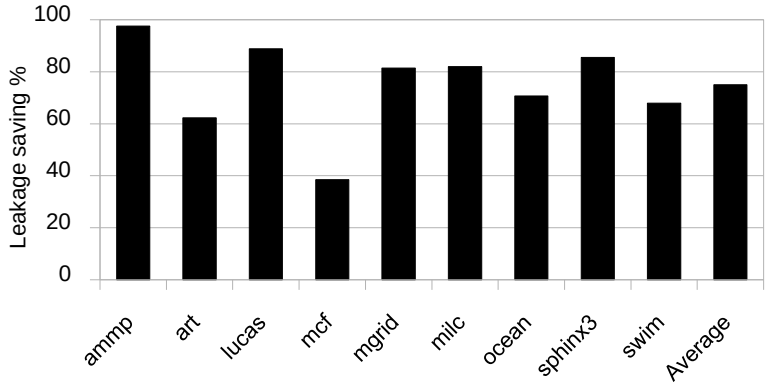


Figure 20: Leakage energy savings.

With dynamic and leakage energy savings combined, the total energy savings PCM-based main memory achieved is 65% averaged over all programs, as shown in Figure 21.

4.6.4.2 Performance Although PCM has slower read and write operations, I found their impact on program performance is quite mild. Similar observations have also been made in [93] where the long latency of the last-level cache implemented using commodity DRAM and LSTP peripheral did not have much impact on the performance of the chip. That is, programs are relatively insensitive to the performance of memory hierarchical levels that are far from the CPU. Note that my platform is 3D stacked chip, meaning that the sensitivity of the performance to the memory latency should be higher than having an off-chip main memory. In other words, if PCM is used as an off-chip main memory, its latency effect on the overall system performance would be even

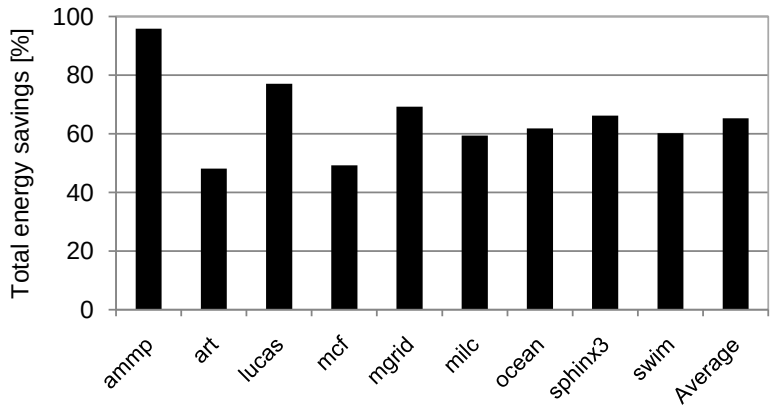


Figure 21: Total energy savings.

smaller, indicating that reducing energy consumption is more important than reducing the latency. My results show that the CPI increase ranges from 0.3% for *ammp* to 27% for *art* with an average of 5.7%, as shown in Figure 22.

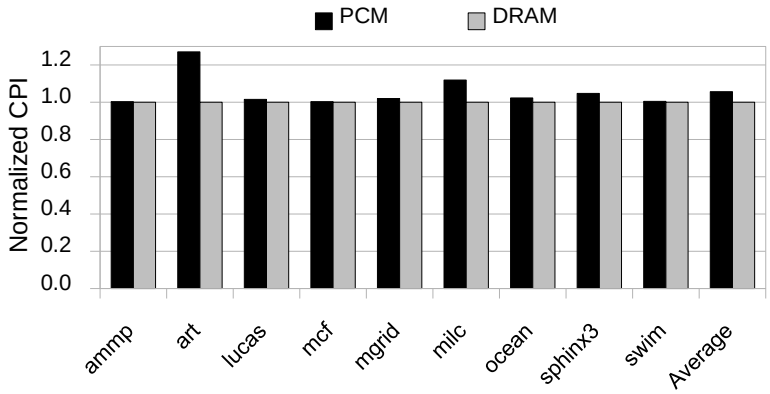


Figure 22: PCM memory latency impact on CPI.

4.6.4.3 Energy-Delay^2 savings I present the results for ED^2 in Figure 23. Due to the great savings in the total energy, and mild increase in execution time, the ED^2 values all showed positive savings, ranging from 96% for *ammp* to 16% for *art*. The average savings achieved is 60%. These results show that using PCM-based main memory in 3D stacked architectures has a great advantage in achieving energy-efficiency.

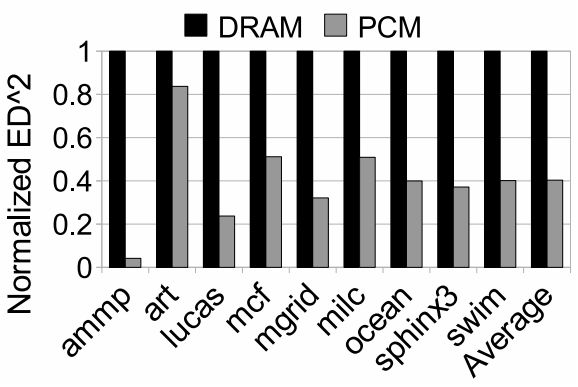


Figure 23: Energy-Delay^2 of PCM memory normalized to DRAM.

4.7 REMARKS

My Differential Write technique and architectural modeling of PCM memory are among the first attempts to use PCM in main memory [103]. Differential Write is not only an effective technique to improve the lifetime of PCM memory, but also opens new opportunities for research in upper level architecture design (e.g., the memory controller) as I will present in the following chapters. The idea of Differential Write was also applied to STT-RAM cache (with a different implementation) to reduce its energy consumption [104]. Several other wear-leveling techniques were proposed for PCM later, each focusing on different goals such as lower overhead or better resilience against malicious programs [77, 79, 90].

5.0 THROUGHPUT IMPROVEMENT OF PCM MEMORY

Although Differential Write is beneficial for endurance and energy, it does not help a lot in PCM's write latency. A PCM write still takes long time to finish, unless every bit in the write request is redundant. In a conventional memory bank design (in which each bank can serve one request a time), this means an active PCM write could block subsequent requests for long time, hurting both latency and throughput. For example, a recent PCM prototype reported by Villa et al. can only achieve 9MB/s write throughput per chip [95]. This could be a big concern as throughput is one of the most important considerations for main memory. Existing schemes such as write-cancelation or write-pausing helps reducing read latency, but does not help throughput. In this chapter, I propose my memory scheduling enhancements at the memory controller level to improve the throughput of PCM memory [105] (Figure 24).

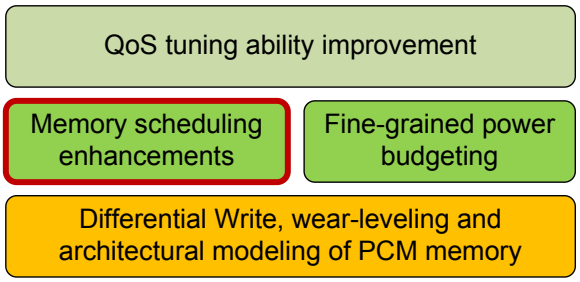


Figure 24: Overview of my research – memory scheduling enhancements.

The novelty of my memory scheduling enhancements is: Existing DRAM-based memory scheduler usually prioritizes read requests over write requests or treats them equally. Once memory requests are dispatched by memory scheduler to bank queues, they are then issued in order. My memory scheduling enhancements, on the contrary, further reorder the requests in bank queues by prioritizing write requests over read requests. This non-conventional design is based on the observation that the throughput of PCM memory is mainly limited by write requests. It also relies on a

non-blocking bank design (which will be discussed in following sections) to prioritize write requests without blocking the read requests for a long time. As I will show in my experiment results, my memory scheduling enhancements can achieve 61% throughput improvement over a baseline bank design.

5.1 BASELINE ARCHITECTURE

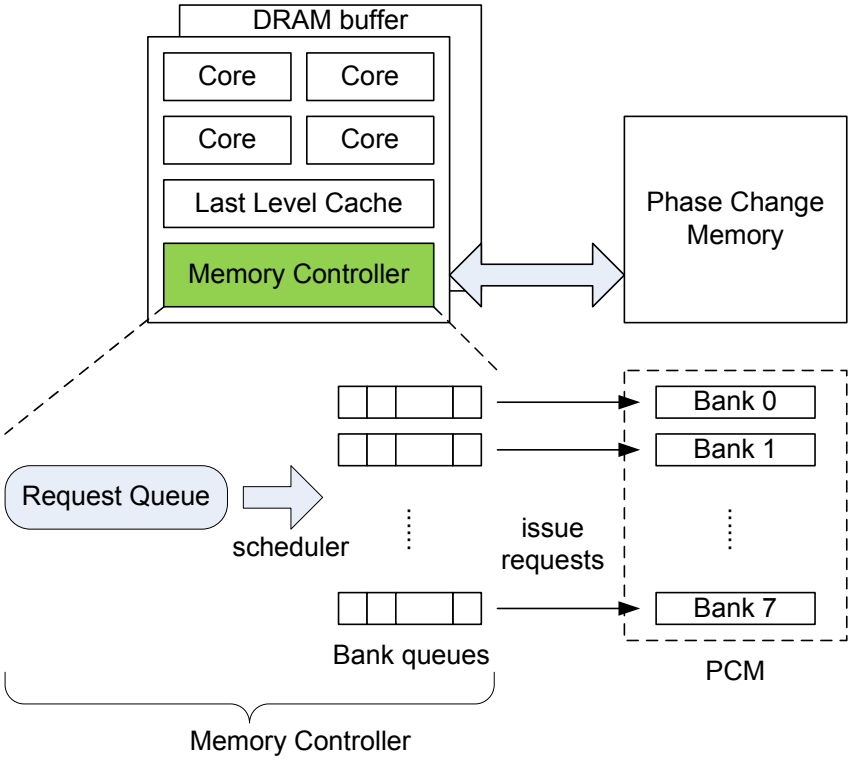


Figure 25: Baseline architecture.

The baseline architecture used in my study is a typical 4-core CMP with private L1 cache and shared L2 cache (as shown in Figure 25). A 128MB DRAM buffer resides between the processor and PCM main memory to mitigate PCM's long latency and limited write endurance. The 4GB PCM memory is organized in a similar way as 8× DRAM DIMM [59] (8 banks per chip, 8 chips per DIMM, single channel DDR2-800 interface). Each memory request is 64-bytes wide, which is the same as the L2 cache block size. Like in DRAM, the PCM memory uses a banking technique

to allow concurrent accesses. In my baseline architecture, I assume the PCM memory supports 8 concurrent accesses. Hence the PCM memory can be viewed as having 8 *logic* banks, each serves requests independently and is indexed by bits 8∼10 of the memory address.

Memory requests arriving at the memory controller are first buffered in a request queue, waiting to be dispatched by the memory scheduler. The scheduler dispatches memory requests to logic banks according to certain policy, typically to maximize bank-level parallelism. In my baseline architecture, I adopt the PAR-BS [63] as the baseline scheduler.

The memory controller also has a row buffer for each logic bank. The row buffers in PCM memory are organized differently than DRAM. Previous research has shown that it is more advantageous to use multiple narrow row buffers for PCM banks [48], since writing a wide row buffer incurs high dynamic power. I use the same architecture in this study: each logic bank's row buffer has 8 256B *row buffer entries*, managed using LRU policy. The total capacity of each row buffer is 8×256B=2KB, which is comparable to a conventional DRAM row buffer. This multiple narrow row buffer entry design is also natural for concurrent intra-bank reads and writes.

My multi-entry row buffer uses a write-through policy, i.e., writing into a logic bank and the corresponding row buffer entry occurs simultaneously. This is because, if I use a write-back policy, a read operation may be held for long time due to slow write back, harming the average read latency. Although write-back policy may reduce the total number of PCM writes, I found this advantage to be diminished: Each 256B row buffer entry contains 4 64B lines, meaning that each write back of an entry would generate at most 4 PCM writes. If each line inside a row buffer entry gets only one write hit, the writing back of this entry would not reduce the total number of PCM writes. Hence a write-back row buffer can reduce total number of PCM writes only when it gets multiple write hits on a same line inside a row buffer entry. However, this opportunity is found to be diminished due to the narrow row buffer entries and the cache/buffer that sits above the PCM memory In fact, no write hits on the same line was observed in my experiments.

5.2 NON-BLOCKING BANK DESIGN

As discussed in the previous section, a logic bank can serve one request a time in the baseline architecture. Due to PCM's long write latency, a request at the head of the bank queue may wait

a long time if the bank is actively serving a write request. This severely impacts PCM memory's throughput, even though PCM has comparable read speed as DRAM. Hence, it is natural to consider parallelizing more concurrent writes in each logic bank to alleviate the problem.

My scheduling enhancements are based on a novel PCM memory bank design, Non-Blocking Bank Design, which allows each logic bank to serve up to two reads and two writes simultaneously (with some restrictions). [1] Non-blocking bank design has an area overhead of 5% (a very conservative estimation), which is much more lightweight than simply having more banks (whose area overhead is estimated to be 10.9%). The basic concept of non-blocking bank is that each logic bank is divided into two "halves", each can be regarded as having 4×8 "regions" (as shown in Figure 26). Each half of a bank can serve one read and one write concurrently, provided that these two requests do not fall into the regions that are on the same "column". As illustrated in Figure 26, when a region is active (in white), regions on the same column of the matrix (in shadow) become unavailable. If a request falls into an unavailable (or active region), it must wait until its region becomes idle. In this thesis, I term this request to be *conflicting* with the on-going request.

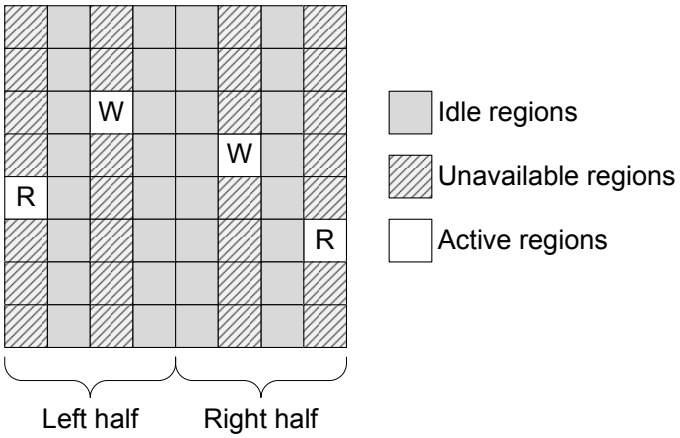


Figure 26: Conceptual view of non-blocking PCM memory bank design.

In summary, there are 3 types of conflicts in this Non-Blocking Bank Design:

- **Write-Write Conflict.** Each half of a logic bank can serve one write a time. Hence if two writes fall into same half of a logic bank, they form a "Write-Write Conflict" and cannot be parallelized.

[1] The non-blocking PCM bank design was created by Bo Zhao (boz7@pitt.edu) at ECE department, University of Pittsburgh.

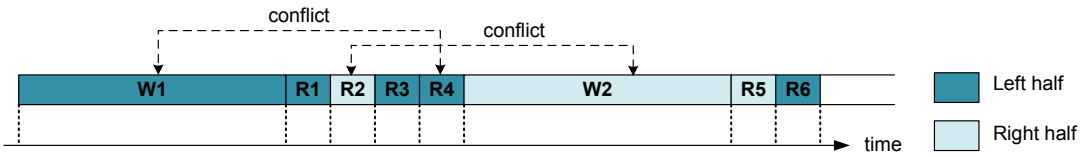
- **Read-Read Conflict.** Similar to the previous case, if two reads fall into the same half of a logic bank, they form a "Read-Read Conflict" and cannot be parallelized.
- **Read-Write Conflict.** As discussed above, each half of a logic bank can serve one read and one write concurrently, provided that they do not fall into the regions that are on the same column. Hence if a half is serving a read request, a write request that falls into the region that is on the same column as the on-going read must wait for the read to finish, and vice versa. This is termed "Read-Write Conflict".

As we can see, this new bank design provides a new opportunity for **intra-bank parallelism** for throughput improvements. And since PCM's throughput challenge is mainly caused by its long write latency, this new bank design has much greater potential than previous Read-While-Write (RWW) designs due to its ability of parallelizing two writes in each logic bank. However, the final throughput improvement also depends on whether the scheduler can fully take advantage of this new opportunity. My experiments have shown that directly applying this new bank design without any scheduling modification only achieves limited throughput improvement. Therefore, enhancements to memory scheduling are needed to exploit the new opportunity.

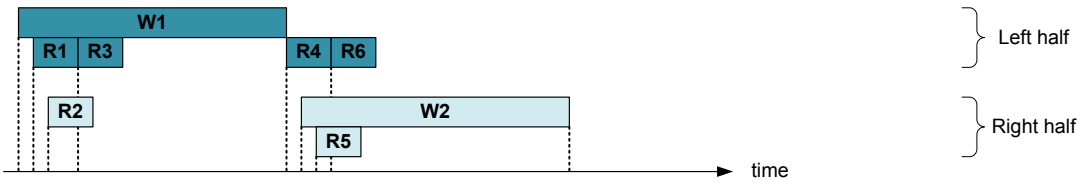
5.3 INTRA-BANK REORDERING: A MOTIVATING EXAMPLE

Non-blocking bank design provides more concurrency inside each bank. However, in many cases this intra-bank parallelism is not utilized if the requests in the bank queue are issued in order. This is mainly due to the conflicts between requests in the bank queue. Consider a bank queue containing the request sequence {W1, R1, R2, R3, R4, W2, R5, R6} (Figure 27(a)). Among these requests, {W1, R1, R3, R4, R6} access the left half, and {W2, R2, R5} access the right half of the logic bank. W1 conflicts with R4 and R2 conflicts with W2. I assume 1000ns and 50ns for write and read respectively [67].

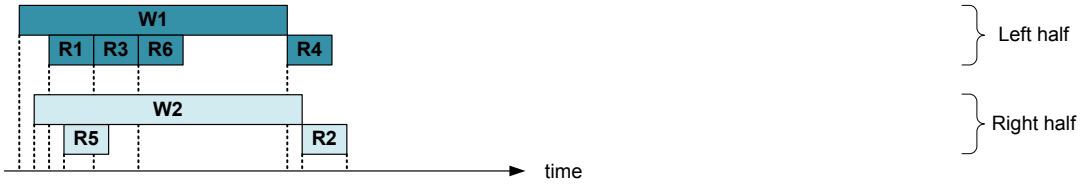
- **Figure 27(a)**. In the baseline (blocking) bank design, each bank can only serve one request at a time. Requests are issued in order. Total time to finish the sequence is ∼2300ns.
- **Figure 27(b)**. We use non-blocking bank design without any scheduling enhancement. Requests in the bank queue are still issued in order. Since R4 conflicts with W1, it cannot be issued until W1 finishes. This delays all subsequent requests, including W2 which could have



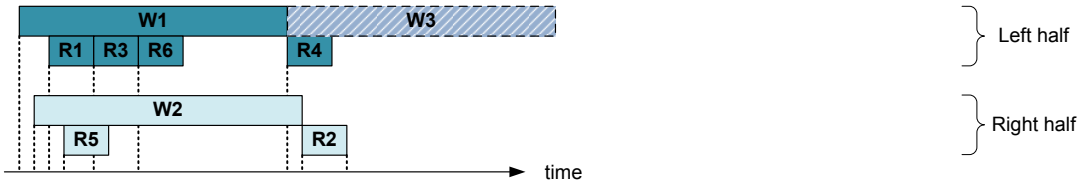
(a) Baseline (blocking) bank design.



(b) Non-blocking bank design, no reordering.



(c) Non-blocking bank design, requests reordered.



(d) Pushing back R4 makes it possible to parallelize with another write (W3).

Figure 27: The impact of intra-bank reordering on request completion time. Assume W1 conflicts with R4 and R2 conflicts with W2.

been served in parallel with W1. Also, it might be desirable to issue R6 sooner (than R4) as it does not conflict with W1. Nevertheless, the total completion time is approximately 1000ns (W1) + 1000ns (W2) = 2000ns, a 13% improvement over the baseline.

- **Figure 27(c)**. We reorder the requests in the bank queue to exploit intra-bank parallelism, assuming dependencies among them have been resolved earlier. In this sequence, W1 and W2 are parallelized. All read requests except R2 and R4 are parallelized with writes. The total time spent by this sequence is approximately 1000ns (W1 and W2) + 50ns (R2 and R4) = 1050ns. Comparing to the baseline, the completion time is reduced by more than 54%.

A key point shown in this example is that in the new non-blocking bank design, reordering requests in the bank queue can effectively improve overall throughput. Also, due to the significant gap between read and write latency, it is important to overlap writes as much as possible to shorten the total latency of the entire sequence. This often requires us to move writes ahead of many reads. But such moves will not hurt the reads too much because they can be parallelized with writes most of the time. If a read conflicts with a write, then I prefer to issue the write first, since the read may have a chance to overlap with another write. For example, in Figure 27(d), R4 follows W1 since R4 may be able to overlap with another write W3 (e.g., another write request in the bank queue). Scheduling R4 before W1 would lose such opportunity for more parallelism unless W3 is also moved ahead of W1 to run concurrently with R4. Either case shows that a write-precedence scheduling policy generates more parallelism. This is also confirmed by my experiments in Section 5.8.2: Comparing the scheme of putting read requests first and my proposed write-precedence scheme, the latter achieves 15% more throughput improvement on average. Therefore, I develop several enhancements of *intra-bank reordering* that usually favors writes over reads to exploit intra-bank parallelism. These are non-conventional, as most existing DRAM-based policies prioritize reads over writes. My algorithm is thus designed for the unique properties of PCM operations.

Since my enhancements target requests inside bank queues to exploit intra-bank parallelism, they can be easily integrated with existing schedulers that target requests inside the memory input queue (exploiting inter-bank parallelism). I use PAR-BS [63] as the inter-bank scheduler, followed by my proposed scheduling enhancements as described below.

5.4 OVERVIEW OF INTRA-BANK REORDERING

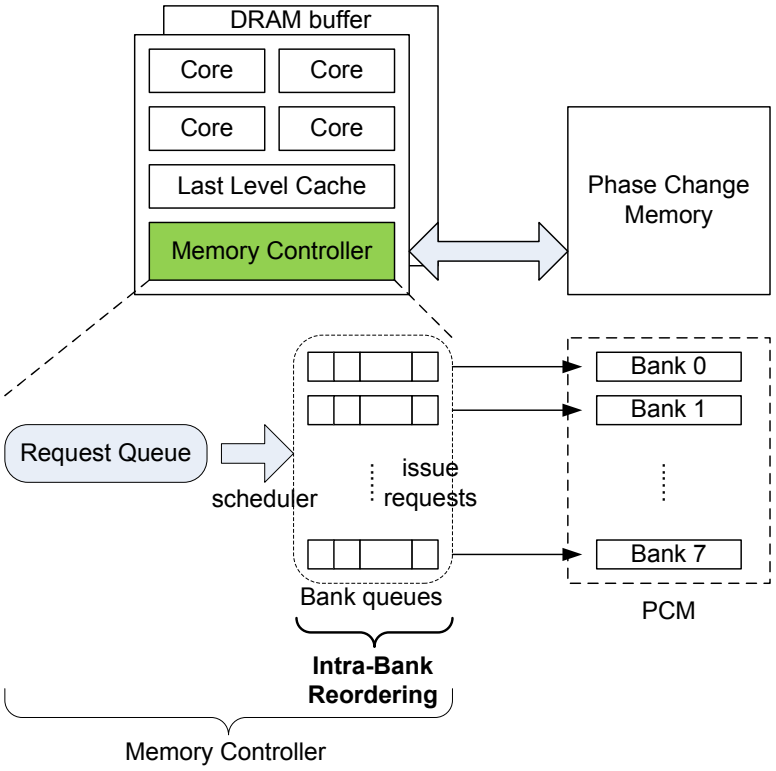


Figure 28: Integration of Intra-Bank Reordering.

My intra-bank reordering enhancements are designed as *extensions* to existing memory scheduler like PAR-BS. As illustrated in Figure 25, the memory scheduler is responsible for dispatching memory requests to individual bank queues under certain policies. My intra-bank reordering enhancements take place in the bank queues after dispatching, as shown in Figure 28. This decoupled design allows my enhancements to work with existing memory schedulers without complex changes to the memory controller. It also enables them to inherit some key scheduling mechanisms from the memory scheduler. For example, the Aggressive Write-Precedence Reordering (AWP) enhancement described in Section 5.5 naturally inherits the batching mechanism from PAR-BS, which is crucial to avoiding starvation. And the Row-Hit Aware Write-Precedence Reordering (RAWP) enhancement (Section 5.6) takes a further step by preserving much of the read row hits promoted by PAR-BS.

5.5 AGGRESSIVE WRITE-PRECEDENCE REORDERING (AWP)

The initial algorithm is directly derived from the intuition developed above. I term this algorithm aggressive write-precedence (AWP) reordering since it aggressively reorders requests in bank queues. The basic idea is to move requests that can be parallelized with on-going requests to the front of a bank queue, so that they can be issued right away.

With non-blocking bank design, one logic bank can serve up to 4 requests at the same time (2 writes and 2 reads), which are termed *slots* of the bank in the following discussion. AWP always tries to fill free slots with suitable (non-conflicting) requests. The algorithm can be described as follows:

Algorithm 1 The AWP algorithm.

```
for all free slot (start with free write slots if available) do
    Look for a request that does not conflict with any on-going or selected requests;
    if request found then
        Mark the request as selected;
    end if
end for
if one or more requests selected then
    Move the selected requests to the front of bank queue, in the order that they were selected;
end if
```

When there are both free write slot(s) and free read slot(s), AWP will try to fill write slot(s) first as shown in Algorithm 1. This means that writes are prioritized over reads, hence the term "write-precedence".

5.5.1 Working with PAR-BS

Since my scheduling algorithm gives priority to writes over reads, there is a chance for a read to be pushed back infinitely. The PAR-BS [63] I use already avoids starvation through request batching: all requests in the previous batch must be issued to the bank before a new batch starts. AWP follows this by reordering only among those requests that are in one batch. Therefore, the starvation avoidance mechanism of PAR-BS is naturally inherited by AWP (as well as my other enhancements).

The original PAR-BS marks up to N requests per thread per bank when creating a new batch. This implies that there might be an imbalanced number of requests for each half bank in a batch, which could harm the parallelism between the two halves. I made a simple revision to PAR-BS by marking up to N/2 requests per thread per *half* bank to balance the requests dispatched to the two halves. I term this simple revision `PAR-BS/Half`. As we will see in Section 5.8, this simple optimization improves throughput over the original PAR-BS by about 30%.

5.5.2 Problems with AWP

With aggressive request reordering in the bank queue, AWP introduces a severe side effect: reduced row buffer hit rate. When requests are dispatched by the memory scheduler (PAR-BS in my case), they are typically ordered to achieve high row buffer hit rate, in addition to improving memory throughput. AWP clearly destroys such locality. My experiments show that the read row buffer hit rate is degraded by up to 65% using AWP (as shown in Section 5.8). Although throughput is important for main memory, the row buffer hit rate affects read access latency, and hence the performance of a workload. Next, I will develop an improved AWP that does not harm the row buffer hit rate that much, but still achieves high memory throughput.

5.6 ROW-HIT AWARE WRITE-PRECEDENCE REORDERING (RAWP)

To overcome the problem of AWP, I develop an improved reordering scheme that maintains row buffer hits while still achieving high throughput – Row-Hit Aware Write-Precedence Reordering (RAWP). To achieve both goals, RAWP follows two principles when reordering requests in a bank queue:

1. Reorder write requests in a similar way as in AWP to achieve high throughput.
2. Issue read requests in similar way as in PAR-BS to maintain row buffer hits.

5.6.1 Read Insertion

The first challenge of RAWP is that when issuing read requests in a similar order as in PAR-BS, some of them may conflict with on-going writes and be blocked for a long time. To address this challenge, I develop a technique called *Read Insertion* to resolve the conflict and preserve the row buffer hit.

Due to PCM cell's high write power and write current, a write access is typically completed in several *rounds* of partial writes [40, 50]. For example, a 512-bit write can be divided into 8 rounds, each round finishing 64 bits. Read Insertion simply allows a read to be "inserted" between two rounds of an on-going write (Figure 29). The on-going write is paused and necessary information is saved so that it can resume properly. The hardware requirement is analogous to the "write pausing" technique [76] which pauses a write for a multi-level cell (which has a different write mechanism from a single-level cell) and resumes it later.

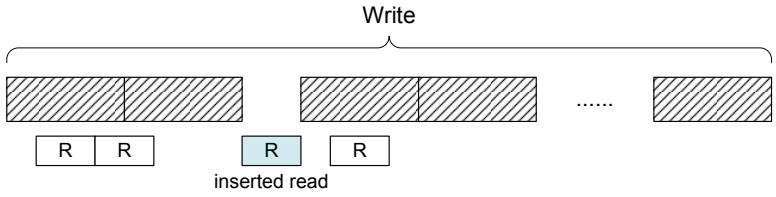


Figure 29: Read Insertion.

With Read Insertion, I can arrange the read sequence in a similar order as in PAR-BS to preserve row buffer hit rate. If a read in the sequence conflicts with an on-going write, it can be inserted in the middle of the on-going write instead of being reordered. Note that Read Insertion

may impact the memory throughput as some parallelizable reads give way to conflicting reads for row buffer hits. As shown in Figure 29, inserting reads increases the latency of the on-going write request, which negatively impacts throughput. However, as I will show in Section 5.8, such an impact is quite limited.

5.6.2 The RAWP Algorithm

With read insertion, RAWP uses a two-step approach to reorder the requests in a bank queue: 1) select issue candidates from the bank queue; 2) form an "issue group" from the candidates in 1) and move them to the head of the bank queue.

Step 1. Selecting issue candidates. In this step, RAWP picks candidates for each free "slot" of the bank. It first ranks the requests. The request with the highest rank will be marked as the issue candidate. Like in AWP which tries to fill write slot(s) first, RAWP starts ranking with free write slot(s). For a write slot, the ranking criteria with decreasing weight are: 1) batched; 2) row hit; 3) thread load; 4) number of reads this write conflicts with, the fewer the better. The information of first 3 criteria are checked and set by PAR-BS. Then for each free read slot, the criteria are: 1) batched; 2) row hit; 3) not conflicting with on-going writes or any write candidates that have already been marked; 4) thread load (information of criteria 1, 2, 4 are checked and set by PAR-BS). For example, a read satisfying the first 3 criteria will have a higher rank than one satisfying first 2 criteria. With the Read Insertion technique, a read that conflicts with an on-going write may still be regarded as "non-conflicting" if the next insertion point of the write is within TH_{RdIns} cycles. In my experiments, this threshold is 20 memory cycles (50ns). The procedure of selecting issue candidates is described in Algorithm 2.

Once candidates are selected for all free slots, they form an "issue group" that will be moved to the head of bank queue. My next step is to determine the relative order among these candidates before moving them to the head of bank queue.

Step 2. Forming an issue group from issue candidates. In this step, RAWP forms an issue group from the candidates picked in the previous step and moves them to the head of bank queue. Depending on their types and row buffer hit status (the $RowHit$ bit set by PAR-BS), the candidates can be classified into four categories: 1) write row hit; 2) write row miss; 3) read row hit

Algorithm 2 RAWP selecting issue candidates from a bank queue

```
for all free write slots do
    for all write requests that can be served by this slot do
        calculate its rank;
    end for
    mark the write request with highest rank as "write candidate" for the write slot;
end for
for all free read slots do
    for all read requests that can be served by this slot do
        calculate its rank;
    end for
    mark the read request with highest rank as "read candidate" for the read slot;
end for
```

and 4) read row miss. RAWP determines the relative order of these candidates according to their categories. In implementation, this can be done by treating the issue group as an "issue queue" and appending candidates to the queue incrementally (starting from an empty queue), as described in Algorithm 3.

RAWP places write hits in the first place since they are beneficial to both throughput and row buffer hit rate. Between read row hits and write row misses, I prefer read row hits because reads are very fast, and they may be parallelized with previous writes. So a subsequent write does not need to wait long. In addition, if we let a write row miss candidate go first, then it may evict a row buffer which may cause the original read row hit candidate to become a miss. Finally, if a read row miss is selected as a candidate, it is not included in the issue group because it does not help throughput, nor row hit rate. Therefore, read row miss candidate(s) will not be moved to the head of the bank queue. Instead, they will will stay at where they were (which was determined by PAR-BS), and will be executed in the order determined by PAR-BS. Once an issue group is formed, all requests must be issued before forming a new issue group.

An example. I now use an example to finish the discussion of the RAWP algorithm. Suppose there is a sequence of requests $\hat{W}_1^*,\ R_2,\ \hat{R}_3^*,\ \hat{R}_4,\ W_5,\ R_6^*,\ \hat{R}_7,\ R_8$ ($\hat{}$ means a request to left half, * indicates a row buffer hit currently). Assuming that all slots are available, and suppose RAWP picks one candidate for each slot: $\{\hat{W}_1^*,\ \hat{R}_3^*\}$ and $\{W_5,\ R_6^*\}$ for left and right half respectively.

Algorithm 3 RAWP forming issue group

Start from an empty issue group (queue);
for all Write row hit candidate(s) **do**
  Append the candidate to the end of issue group (queue);
end for
for all Read row hit candidate(s) **do**
  Append the candidate to the end of issue group (queue);
end for
for all Write row miss candidate(s) **do**
  Append the candidate to the end of issue group (queue);
end for
Move all candidates to head of bank queue, in the order as they are in the issue group (queue);

Next, RAWP forms an issue group from these requests following Algorithm 3: $\hat{W}_1^*$, $\hat{R}_3^*$, R_6^*, W_5. This is because $\hat{W}_1^*$, $\hat{R}_3^*$ and R_6^* are all row hits, and W_5 is a miss. As we can see, all row hit requests are preserved in this issue group, illustrating the effectiveness of RAWP. Moreover, after this issue group, the row hit/miss status will change since the write miss will update the row buffer. Hence, there might be new row hits for the remaining reads because of this change. This is the reason why sometimes RAWP can result in a little higher row buffer hit rate than the original PAR-BS.

5.7 IMPLEMENTATION AND OVERHEAD CONSIDERATIONS

Due to the complexity of PAR-BS algorithm, my baseline scheduler is likely to be implemented with a microcontroller running a firmware. My intra-bank reordering techniques (AWP and RAWP) are enhancements to the memory scheduler. Therefore they can be implemented as additional subroutines in the PAR-BS firmware.

The computation overhead of AWP and RAWP is relatively small compared to the PAR-BS algorithm. For example, PAR-BS needs to scan the request queue (up to 1024 entries) when forming a new batch. It also uses a complex ranking system when dispatching the requests. AWP and RAWP, on the other hand, only need to scan small bank queues (up to 32 entries). During an intra-bank reordering procedure, AWP only needs to check conflicts. RAWP needs to rank the

requests to determine the issue candidates, but it reuses most of the ranking information from PAR-BS. Moreover, AWP and RAWP are enhancements to memory scheduler, so they are optional steps in memory scheduling and are not on the critical path. Memory scheduler may delay or skip the intra-bank reordering step when it is busy.

5.8 EVALUATION

5.8.1 Experimental Setup

I experimented with various hardware designs (baseline blocking design, Read-While-Write design, proposed non-blocking design) and scheduling enhancements (AWP, RAWP). The notations and explanations are listed in Table 5. [2]

Table 5: Notations of bank designs and scheduling schemes used in experiments.

Notations of Bank Designs	
RWW	Read-While-Write hardware, 1 read, 1 write per logic bank
NB	Non-Blocking Bank Design, 2 reads, 2 writes per logic bank
Notations of Scheduling Enhancements	
PAR-BS/Half	PAR-BS with half-bank marking, no intra-bank re-ordering
AWP	PAR-BS/Half enhanced with Aggressive Write-Precedence reordering
RAWP	PAR-BS/Half enhanced with Row-hit Aware Write-Precedence reordering

I collected memory access traces and fed them to a detailed memory model to measure the memory throughput. I ran memory intensive benchmarks from SPEC2006, SPLASH2 [88], and STREAM [58] on a 4-core CMP simulator in Simics [56]. Each core runs at 3.2GHZ with 4MB shared L2 cache, a 128MB DRAM buffer and 4GB main memory. The memory module I used was developed in the GEMS [57] framework. I heavily modified the memory controller module to model more details such as bank/half busy counters, bus contention, row decoder contention, column conflicts, channel contention, etc. I then implemented my non-blocking reordering schemes based on this model. Further parameters are listed in Table 6.

[2]The non-blocking bank design is developed by Bo Zhao (boz6@pitt.edu) at ECE department, University of Pittsburgh.

Table 6: Parameter used in experiments.

Platform	
Cache	32K private L1 cache, 4MB shared L2 cache, 64-byte cache lines
PCM settings	
Interface	Single DIMM, 8 chips, 8 banks, single channel DDR2-800 interface, 6.4GB/s peak bandwidth
Read	50ns (row buffer miss) / 10ns (row buffer hit)
Write	8–round setting: 1000ns (200ns + 8 rounds × 100ns) 4–round setting: 600ns (200ns + 4 rounds × 100ns)
Row Buffer	Multiple-entry row buffer, 8 × 256-byte entries

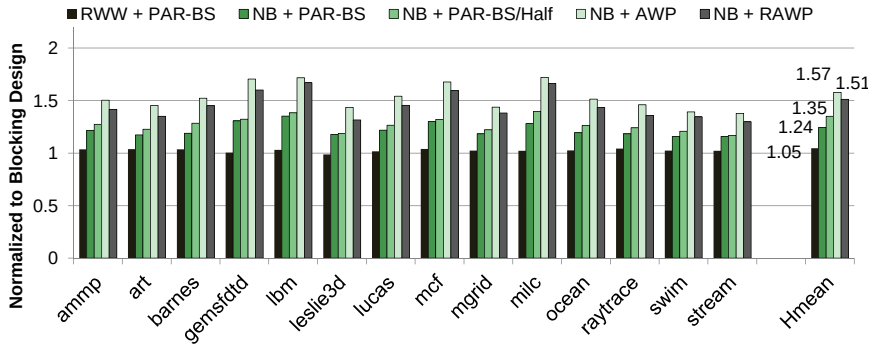
The parameters adopted in PCM write have significant impact on overall throughput and the effectiveness of scheduling policies. For example, PCM write latency affects how much throughput improvement I can get, and the number of "write rounds" in each PCM write determines the number of "insertion points" that Read Insertion can exploit. For completeness, I experimented with both 8-round PCM write (which is the default setting) and 4-round PCM write configurations when comparing the throughput of different scheduling schemes.

5.8.2 Throughput Improvement

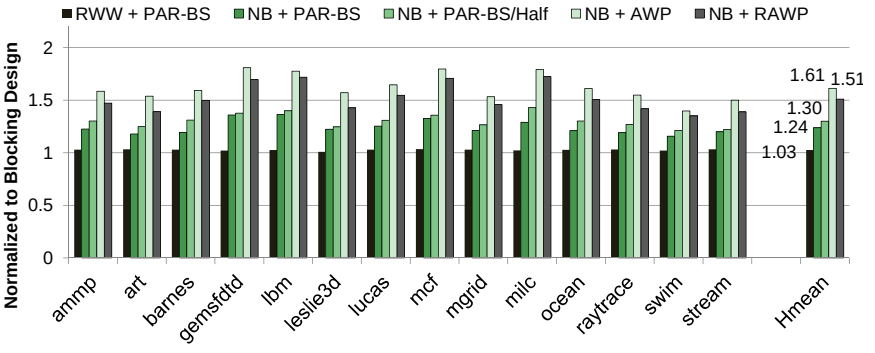
The first set of results I present is memory throughput ($Number\_of\_Requests\_Served/Total\_Time$) improvement for various schemes. Figure 30(a) and 30(b) show results for 4-round and 8-round settings respectively. The results are normalized to blocking design that also uses PAR-BS scheduler.

As we can see, `RWW` achieves 3∼5% throughput improvement on average because of its limited hardware capability. Using `NB` without any scheduling enhancement (NB + PAR-BS) improves throughput to 24% because my hardware provides much more parallelism. Although the proposed hardware design is quite effective, much more parallelism can be achieved through scheduling enhancement. `PAR-BS/Half` does a simple change to balance batch size between two halves inside each bank, resulting 30∼35% throughput increase. `AWP` achieves the highest improvement (57∼61%

on average) because of its aggressive reordering algorithm. RAWP not surprisingly has smaller improvement (51%) than AWP because of a carefully crafted algorithm that preserves row buffer hit rate.



(a) 4-round write configuration



(b) 8-round write configuration

Figure 30: Throughput improvement.

As I have discussed in Section 5.3, AWP often prioritizes write requests over read requests. In Figure 31, I compare the throughput improvement between NB+AWP and a scheme that puts read requests first (NB+RP). On average, my proposed write-precedence scheme achieves 15% more throughput improvement over the baseline. These results back my motivation of applying write-precedence reordering on non-blocking bank design.

5.8.3 Preserving Row Buffer Hits

As discussed earlier, RAWP overcomes the shortcomings of AWP and preserves row buffer hit rates. Figure 32 shows the read row hit rates for AWP and RAWP respectively. The results are normalized to non-blocking circuit with PAR-BS (NB + PAR-BS). AWP dramatically degrades

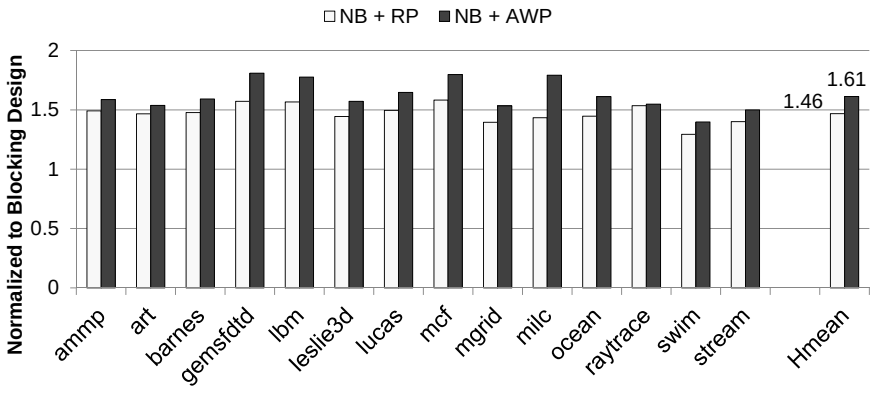


Figure 31: Why write-precedence: comparing throughput improvements with a scheme that puts read requests first (NB+RP).

row buffer hit rate by up to 65% (35% on average) due to its aggressive reordering design. On contrary, RAWP preserves row buffer hit while achieving a comparable throughput improvement to AWP.

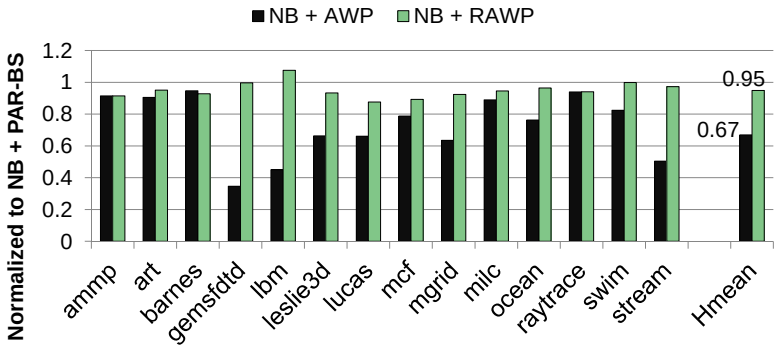


Figure 32: Read row hit rate under different scheduling enhancements.

5.8.4 Performance

I estimated system performance improvements under different scheduling enhancements, as shown in Figure 33. The performance metric I used is Weighted Speedup [89], which is commonly used to measure the performance improvement for multiprogrammed systems. Weighted Speedup is calculated as follows:

$$Weighted\_Speedup = \sum_i \frac{IPC_i^{shared}}{IPC_i^{alone}}$$

IPC_i^{shared} and IPC_i^{alone} are the instruction-per-cycle metrics of each thread when running along with other threads and when running alone respectively.

As shown in Figure 33, using non-blocking bank design without any intra-bank reordering (`NB+PAR-BS` and `NB+PAR-BS/Half`) results in 5.9/5.2% improvement. With scheduling enhancements, AWP and RAWP achieve 9.5% and 9.6% improvement respectively. Although AWP can achieve higher throughput improvement, its low read row buffer hit rate hurts read latency. As a result, AWP achieves slightly lower performance improvement than RAWP on average.

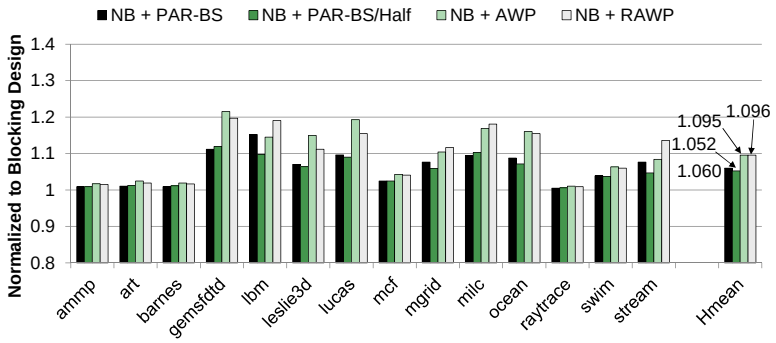


Figure 33: Weighted Speedup [89] for different scheduling enhancements.

5.8.5 Comparing with Write Cancelation and Write Pausing

I compare the performance (weighted speedup) of RAWP with Write-Cancelation and Write Pausing [76]. When a read is blocked by an on-going write, write-cancelation allows the write to be canceled and issues the read first. The canceled write request is restarted later, meaning that the time that is already spent is wasted. Write pausing uses a similar idea to optimize read latency, except that the write operation is resumed from the point it was interrupted, saving the time it

already spent. As we can see, both techniques only optimize read latency but not memory throughput. I built these techniques on top of non-blocking bank design and PAR-BS/Half to perform a fair comparison (`NB+WC` and `NB+WP`). The results are shown in Figure 34. The average performance improvement for write-cancelation, write-pausing and RAWP are 4.7%, 5.1% and 9.6% respectively. RAWP aims to produce request parallelism, including read-read parallelism which is particularly helpful in reducing the average read latency. Write-cancelation and write-pausing, however, only preempt a write when a read comes in, so they cannot exploit the benefits produced by parallelism.

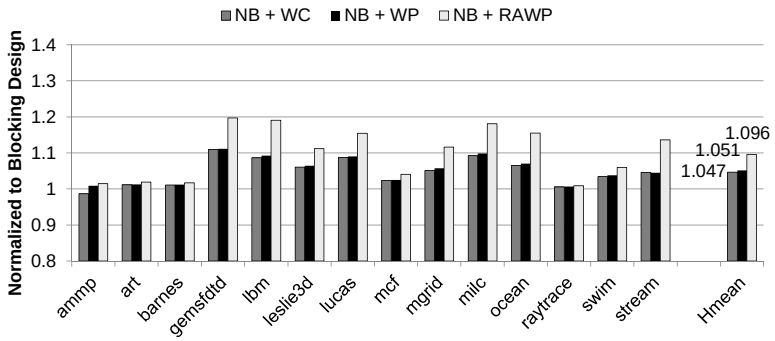


Figure 34: Weighted Speedup [89] for Write Cancelation, Write Pausing [76] and RAWP.

6.0 THROUGHPUT IMPROVEMENT UNDER POWER BUDGETS

In this chapter, I propose my technique for fine-grained power budgeting for PCM memory [105] (Figure 35).

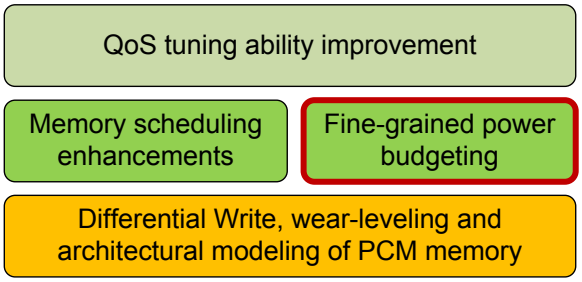


Figure 35: Overview of my research – fine-grained power budgeting.

Due to PCM's high write power and write current, PCM memory typically uses charge pumps in its write drivers [40]. This is expensive and challenging because charge pumps have to supply high current while sustaining high voltage at same time [40]. Moreover, the transient high current generated by multiple concurrent bit writes can cause noise on the power line, which may hurt stability [40]. As a result, practical PCM memory is usually designed with some limitation on the number of concurrent bit writes [15, 40], which is termed its "power budget" here.

For example, the baseline PCM bank discussed in the previous chapter has 8 logic banks, allowing up to 8 concurrent PCM write requests. Assuming each 512-bit write request is completed in 8 rounds (meaning that each round writes 64 bits concurrently), the 8 concurrent write requests generate 8×64 concurrent bit writes. Hence the power budget of the baseline bank design is 8×64=512 concurrent bit writes.

The increased parallelism achieved by non-blocking bank design and my memory scheduling enhancements leads to a larger number of concurrent PCM write requests. For a non-blocking PCM memory also with 8 logic banks, there could be up to 16 concurrent write requests, resulting

in up to 16×64 concurrent bit writes. Apparently, it may exceed the original power budget used by baseline bank design. Due to the reasons I have just discussed, simply raising the power budget of PCM memory is expensive and unfavorable. Hence, a technique is needed to preserve the throughput improvement from my scheduling enhancements without having to raise the power budget.

6.1 OPPORTUNITIES FROM DIFFERENTIAL WRITE

In order to improve PCM memory throughput without raising its power budget, an intuitive approach is to improve the *utilization* of the power budget. This can be accomplished by reducing the number of bit writes in each write request, so that more write requests can be served concurrently under same power budget. The Differential Write (or "DW" in short) technique proposed in Chapter 4 offers a great opportunity here: With DW, about 85% of bit writes can be avoided in PCM write requests. Moreover, Differential Write provides important information that can be leveraged by power budgeting techniques for better estimation of power demands. For instance, the original power budget used by the baseline bank design (8×64 concurrent bit writes) allows only 8 concurrent write requests. After applying Differential Write, many of the write requests will have power demands less than 64 bit writes. So, if the power budgeting technique has the bit change information provided by the Differential Write, more than 8 concurrent write requests can be served at the same time, as long as their total power demand does not exceed 8×64 concurrent bit writes.

Another scheme that can reduce the number of bit writes in each write request is Flip-N-Write (or "FnW" in short) [12]. Unlike DW that writes only those bits that are changed in each write, the FnW technique writes either the original value or its inversion, whichever results in fewer bit flips. Between DW and FnW, DW could result in fewer bit flips than FnW sometimes, but does not have an upper bound on the bits being written. FnW, on the other hand, can guarantee that the number of bits being written is no more than half of the total write width. Both schemes are beneficial for the reduction of power demand in write requests. My experiments will show that incorporating both schemes into my power budgeting technique can yield better results.

6.2 OVERVIEW OF BIT LEVEL POWER BUDGETING

The goal of my power budgeting technique is to achieve better throughput under the same power budget and the same memory scheduler. In order to achieve flexibility, I use a decoupled design in my power budgeting technique (as shown in Figure 36). The key component of my technique is the Power Budgeting Controller (PBC), which is responsible for the power gating of PCM memory. If PBC thinks a write request issued by the memory scheduler will cause a violation of power budget, it will hold this issued request (i.e., it will not activate it) until there is adequate power budget. Note that the power budget is defined as the number of concurrent bit writes. Hence my power budgeting technique only restricts write requests and does not restrict read requests.

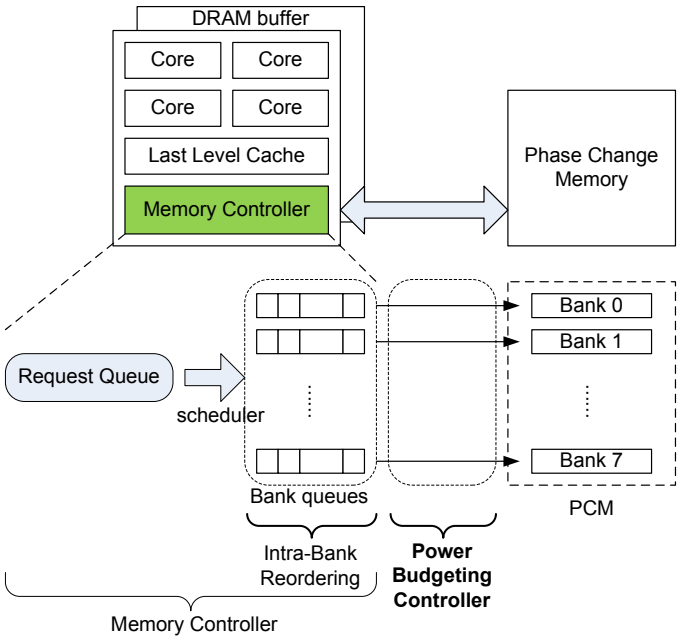
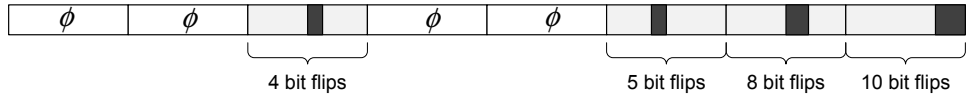


Figure 36: Overview of Power Budgeting enhancement.

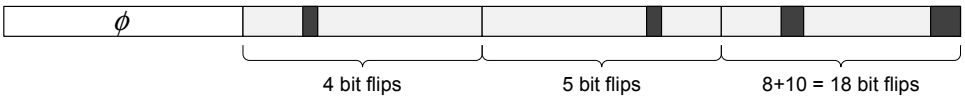
In my proposed BPB design, PBC is extended with the ability of choosing write configurations for issued write requests. For each issued write request, its *write configuration* determines how many rounds it will take to finish. Instead of a fixed write configuration, BPB supports flexible write configurations. For example, a write request may choose an 8-round configuration, meaning that it will finish in 8 rounds (with each round writing 64 bits), or it may pick a 4-round configuration which finishes in 4 rounds (with each round writing 128 bits).

The basic idea of my Bit Level Power Budgeting (BPB) technique is to leverage the fine-grained bit-change information provided by Differential Write to get better estimation of a write request's *power demand*, and to facilitate the decisions of write configuration at runtime.

With Differential Write, the number of bit changes in each write request is no longer fixed. Splitting a write request into 4 rounds or 8 rounds may result in different power demand (due to different widths of each round) and different latency (due to different number of rounds). And depending on the distribution of bit-changes in the write request, power demand of each round could also be different. I assume that the power demand of a write configuration is determined by the round with most number of bit changes, then a 4-round configuration tends to have higher power demand than an 8-round configuration. As illustrated in the example in Figure 37, the 4-round configuration has power demand of 18 bit changes while the 8-round configuration has power demand of 10 bit changes. Under the same power budget, this means the 4-round configuration for this write request may start later because it may be held for a longer time by the PBC. On the other hand, however, the 4-round configuration can result in shorter latency due to its fewer number of rounds. Hence the completion time of the two write configurations is decided by both factors, i.e., the start time (T_{start}) and latency ($T_{latency}$). PBC needs to evaluate this trade-off in order to pick a better configuration. Details of this procedure will be discussed in the following sections.



(a) 8-round configuration



(b) 4-round configuration

Figure 37: Power demand of different write configurations.

In summary, the PBC in my BPB scheme has several enhancements in addition to simple "power gating" tasks:

- Keep track of power consumption at fine granularity (e.g., number of concurrent bit writes);
- Collect fine-grained (bit-level) information of each write request's power demand by leveraging the information from Differential Write (e.g. number of bit changes and their distributions);

- Instead of using a fixed write configuration for all write requests, PBC can assign individual write configuration for each issued write request to improve the overall throughput under the same power budget.

My BPB technique chooses a decoupled design that works independently with the memory scheduler. PBC may throttle some of the issued requests from the memory scheduler, yet it does not change the request order in each bank queue (which is determined by the memory scheduler). The major reason I choose this design is to ensure flexibility and simpleness of integration:

- A decoupled design makes BPB able to easily work with different memory schedulers. System with the existing memory scheduler can add the BPB extension without complex changes in the memory controller firmware. This flexibility makes my BPB technique more feasible for adoption in existing systems.
- The decoupled BPB does not need the knowledge of the memory scheduler, but focuses on the power information of PCM memory. This means the PBC can collect and track information locally on the memory module. On the contrary, a closely-coupled design requires PBC to transmit power information between the memory module and the memory controller, increasing latency and bandwidth overhead.

  In the following sections, I will describe the details of my BPB technique.

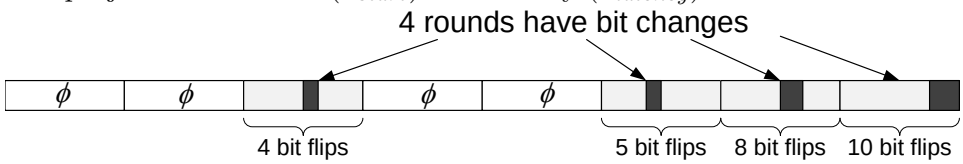
6.3 TRADE-OFFS IN FLEXIBLE WRITE CONFIGURATION

Conceptually, a 64-byte write request is chopped into several parts and written sequentially in several rounds. A write configuration determines how the line is chopped and how many rounds the write request will take to finish. For example, an 8-round configuration means a 64-byte write request is chopped into 8 64-bit parts and will finish in 8 rounds, with each round writing 64 bits. In this thesis, I assume four possible write configurations (8-, 4-, 2- and 1- round), and the 64-byte line is chopped evenly.

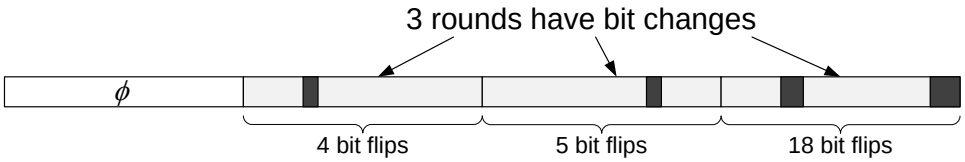
With Differential Write (DW), the number of bit changes in a write request is not fixed. Depending on the distribution of bit changes in the line, the number of of bit changes is also different in each round of the write request. In BPB, I use the round with most number of bit changes to

estimate the power demand of the whole write request. As shown in the example in Figure 37, the 4-round configuration has a power demand of 18 bit writes, and the 8-round configuration has a power demand of 10 bit writes for the same write request.

As we can see from the example, configurations with fewer number of rounds tend to have higher power demands than those with a larger number of rounds. When the available power budget is low, this means the former configurations may be held by PBC for a longer time to wait for enough power budget. On the other hand, fewer number of rounds results in shorter latency. This brings up the trade-off between less power demand and shorter latency: Choosing a configuration with fewer number of rounds can results in shorter latency, but the write request may be held for longer and starts later. Choosing a configuration with a larger number of rounds results in longer latency, but the write request may start earlier due to lower power demand. PBC must evaluate the trade-off to pick a better configuration. Essentially, it should look at the *completion time*, which is the sum of the request's projected start time (T_{start}) and latency ($T_{latency}$).



(a) 8-round configuration.



(b) 4-round configuration.

Figure 38: Comparing different write configurations. Boxes containing ϕ are redundant rounds that can be skipped. Dark parts are bit changes. Those rounds must be performed.

Another flexibility introduced by my BPB scheme is the variable number of rounds by skipping *redundant rounds*. Previous work has shown that on average, 85% of bit writes are redundant in memory [103]. When a line is chopped into several parts and written in several rounds, it is possible that some of the parts are entirely redundant (i.e., do not have any bit changes at all). BPB allows skipping of these "redundant rounds" to reduce the actual number of rounds and latency. Figure 38 shows an example of this technique. In this example, the actual number of rounds in the 8-round configuration is 4, and the actual number of rounds performed in the 4-round configuration is 3.

With this technique, a write configuration determines the upper bound of its number of rounds instead of a fixed one (e.g., an 8-round configuration means a write request will finish in *up to* 8 rounds). Also it can be seen from the example that: Comparing the actual number of rounds performed, the 4-round configuration is only 1 round less than the 8-round configuration, but the 4-round configuration has a much higher power demand. When evaluating the trade-off between less power demand and shorter latency, PBC must also take this new variable into consideration.

6.4 THE BPB ALGORITHM

As I have discussed in the previous section, my BPB algorithm picks a write configuration for each write request by looking at its *completion time*. The criteria can be expressed as:

$$T_{finish} = T_{start} + T_{latency}$$

T_{start} is the time in the future when there is enough power budget for this write configuration. In reality, this number is affected by many other factors in addition to power budget, such as channel busy status, command bus availability, etc. $T_{latency}$ is the time required to serve the write request. This is mainly determined by how many rounds are actually performed. Since my BPB scheme allows skipping redundant rounds, this number is variable depending on the distribution of bit changes. If Read Insertion is enabled, an on-going write request may have several read requests inserted, adding more uncertainty to this value. Hence, it is difficult to calculate the T_{finish} value accurately at runtime. In my scheme, instead of trying to calculate T_{finish} accurately, I use its lower bound – the "earliest possible finish time" – as a simple approximation.

For a particular write configuration, $\hat{T}_{start}$ is defined as its "earliest possible start time". In other words, I try to answer the question: If no new request is activated from now on, when will there be enough power budget for this write configuration? Under this assumption, power consumption of the PCM memory goes down over time as more and more on-going requests finish (as illustrated in Figure 39). By adding a "remaining cycles" counter to each on-going request, I can calculate this curve and determine the earliest possible time when available power budget will be enough for this write configuration.

$\hat{T}_{latency}$ is the "shortest possible" latency for the write configuration. It can be calculated from the number of non-redundant rounds in this write configuration, assuming no read is inserted.

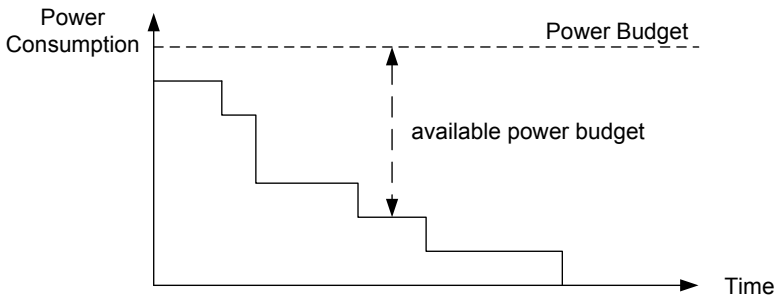


Figure 39: Computing the "earliest possible start time" of a write configuration. Assuming no new requests are activated from now on, power consumption will go down over time as more and more requests finish.

Algorithm 4 BPB algorithm: picking write configuration for a write request.

Initialize $MinT_{finish}$ to a large number;
Initialize $WriteConf$ to the default configuration (8-round configuration);
for all possible write configurations **do**
  Calculate the earliest possible finish time $\hat{T}_{finish}$ for this write configuration;
  Calculate the power demand P_{demand} for this write configuration;
  if $P_{demand} > DemandCap$ **then**
    Skip this configuration;
  else
    if $\hat{T}_{finish} < MinT_{finish}$ **then**
      $MinT_{finish} = \hat{T}_{finish}$;
      $WriteConf$ = current write configuration;
    end if
  end if
end for
Return $WriteConf$ as the picked write configuration;

With $\hat{T}_{start}$ and $\hat{T}_{latency}$, BPB can compute the "earliest possible finish time" ($\hat{T}_{finish}$) for a given write configuration: $\hat{T}_{finish} = \hat{T}_{start} + \hat{T}_{latency}$. When choosing a write configuration for an issue write request, BPB iterates on each possible write configuration and picks the one that generates the minimum $\hat{T}_{finish}$.

The "greedy" nature of this procedure tends to grab as much power budget as it can to reduce latency. This could sometimes lead to a situation that a few write requests with high power demand use up most of the available power budgets and cause other issued requests to be held. This is undesirable as there might be low-demand requests that could be otherwise served in parallel. To avoid this side effect, I add a simple constraint *DemandCap* to the iteration to limit power demand of the picked write configuration. Based on my experiments, I set *DemandCap* to 64 in my evaluation. The revised procedure of selecting a write configuration for a write request is described in Algorithm 4.

6.5 IMPLEMENTATION AND OVERHEAD CONSIDERATIONS

Computing $\hat{T}_{start}$ and $\hat{T}_{latency}$ requires the knowledge of number of changing bits and their distribution in the current write. When a write request is issued, BPB leverages Differential Write to do the pre-write read and compare to obtain such information. During the pre-write read operation, the read circuit is occupied by DW and cannot serve other read requests. This extra blocking time is also modeled in my simulation. BPB then uses the information to pick a write configuration from four possible options (8-round, 4-round, 2-round and 1-round) for the write request.

Choosing write configuration requires the calculation of $\hat{T}_{start}$. A possible implementation to calculate $\hat{T}_{start}$ is to maintain a sorted list, "T-P list", for all on-going write request:

$$< T_0, PD_0 >, < T_1, PD_1 >, < T_2, PD_2 >, ...$$

Where T_i and PD_i are the finish time and power demand (number of bit changes) of each on-going write request. The list is sorted by T_i.

The "T-P list" is maintained incrementally at runtime. Once an issued write request is activated, PBC inserts a new entry into the T-P list for it. In the new entry, the T_i value is the current time plus the latency of the write request, and the PD_i value (power demand) was calculated earlier

when the write request was issued and assigned with a write configuration. When a write request finishes, PBC removes the corresponding entry from the T-P list. In the case of Read Insertion, PBC updates the T_i value of the paused write request.

The size of the T-P list is determined by the number of concurrent write requests. For the 8-bank configuration used in my experiments, the maximum size of T-P list is 16 entries (as there could be up to 16 concurrent writes).

For a given write configuration of a given write request, PBC can use the T-P list to calculate $\hat{T}_{start}$ for it (described in Algorithm 5). Essentially, PBC walks through the T-P list, accumulates the PD_i values and stops when the power demand of the given write configuration is met.

Algorithm 5 BPB algorithm: calculation of $\hat{T}_{start}$.

Suppose currently available power budget is P_{avail} (in number of bit changes);
Suppose power demand of the write configuration is P_{demand};
$P_{cur} = P_{avail}$;
if $P_{cur} \geq P_{demand}$ **then**
  Return current time;
else
  for all entry $< T_i, PD_i >$, start from head of list **do**
    $P_{cur} = P_{cur} + PD_i$;
    if $P_{cur} \geq P_{demand}$ **then**
      Break the loop and return T_i;
    end if
  end for
end if

For a given write request, PBC needs to evaluate $\hat{T}_{start}$ for each of the four write configurations (8-round, 4-round, 2-round and 1-round). This can be further optimized into only one scan of the T-P list. For a same write request, a configuration with fewer number of rounds will always have a higher or equal power demand than a configuration with a larger number of rounds. For example, the power demand of a 4-round configuration will never be lower than the power demand of an 8-round configuration for the same write request. Therefore, scanning of the T-P list (Algorithm 5) for the 4-round configuration will never stop earlier than the 8-round configuration, and so on. I can leverage this "inclusive" property to optimize the procedure into one scan: For the given write request, I only scan the T-P list for the 1-round configuration and remember the additional

$\hat{T}_{start}$ results for the 8-round, 4-round and 2-round configurations during the process. Once the $\hat{T}_{start}$ values are ready, PBC can choose the write configuration for the given write request. This is essentially a process of picking 1 out of (up to) 4 choices.

With T-P list and above optimization, choosing a write configuration for an issued write request is essentially an accumulate-and-comparison loop with up to 16 iterations. Since the write configuration is picked when a write request is issued, and the write request often needs to wait for enough power budget before it can be activated, the time to perform this accumulate-and-comparison loop may overlap with the time the write request waits for its activation. In my experiments, however, I charge a constant 100ns penalty for each write request. My experiment results show that it is still worthwhile to apply BPB to improve throughput under power budget.

6.6 EVALUATION

6.6.1 Experimental Settings

My evaluation of BPB is based on a similar experimental platform as used in the previous chapter (Section 5.8), with my extensions of BPB scheme in the simulator. I assume the write power to be 1.22mW per bit, which is scaled from a Samsung prototype [40]. In addition to the original power budget of baseline bank design (8×64 concurrent bit writes), I also experimented with several other power budget settings (4×64, 12×64 and 16×64 bit writes) for completeness. PAR-BS with RAWP enhancement (discussed in Chapter 5) is used as memory scheduler in my experiments.

Baseline. My baseline scheme is RAWP with simple power gating without Differential Write or Flip-N-Write functions. As described in Section 6.2, PBC estimates power demand for each write request, checks it against current available power budgets and throttles the request if it thinks it will violate the power budget. In the baseline, PBC does not have bit change information from Differential Write, meaning that it assumes every bit will be changed in each round. Baseline also does not have other enhancements like flexible write configurations, skipping redundant rounds as I proposed in BPB.

Besides the baseline, three different schemes are compared in my experiments:

- `FnW-only`, which simply adds Flip-N-Write function to the baseline. With Flip-N-Write (which guarantees that up to half of the bits are changed), PBC always estimates that half of the bits

in each round will be changed when checking against available power budget. Hence it has limited flexibility to choose between 8-round and 4-round configurations. `FnW-only` does not skip redundant rounds as it does not retrieve and process bit change information in PBC.

- `BPB`, which is my Bit Level Power Budgeting scheme using bit change information from Differential Write. In this scheme, PBC leverages the information from Differential Write to estimate power demand for each write request. It supports flexible write configurations and can skip redundant rounds.
- `BPB+FnW`, which is my Bit Level Power Budgeting with a combination of Differential Write and Flip-N-Write. As I discussed earlier, DW can provide fine-grained bit change information, but does not have an upper bound of how many bits are changed. In some cases when write requests have a high number of bit changes, using DW alone could result in high power demand and hurt throughput improvement (as I will show shortly). By combining both techniques with BPB, I can effectively avoid this shortcoming.

6.6.2 Throughput Improvement Under Power Budgets

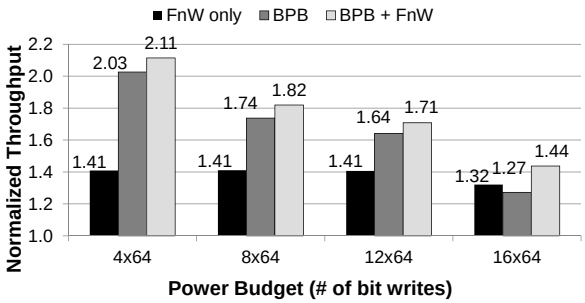
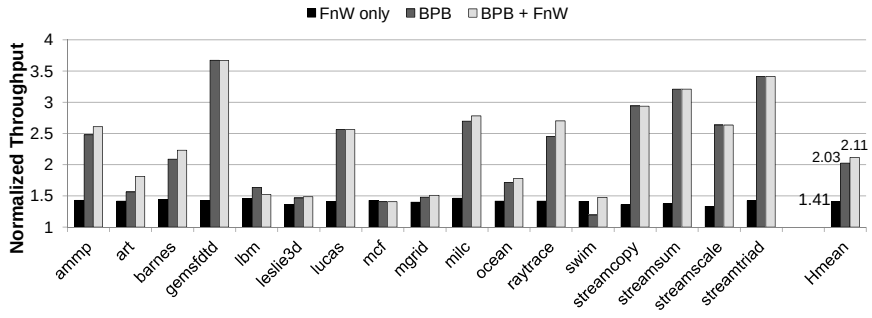


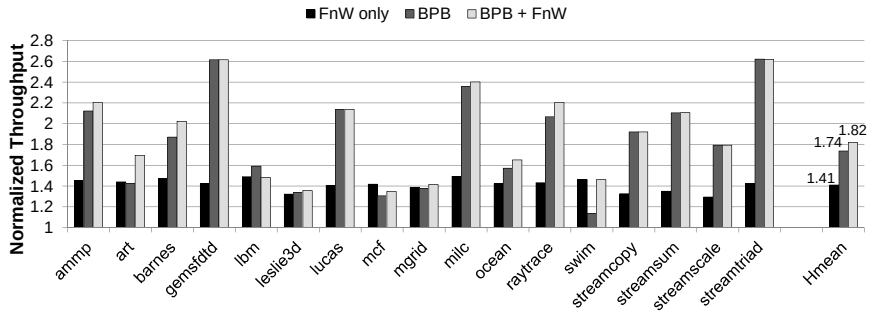
Figure 40: Average throughput under different power budgets, normalized to baseline (RAWP with simple power gating).

I now present throughput improvements for different power budgeting schemes. All results are normalized to the baseline. I summarize the average results for different power budgets in Figure 40, and detailed results are presented in Figure 41.

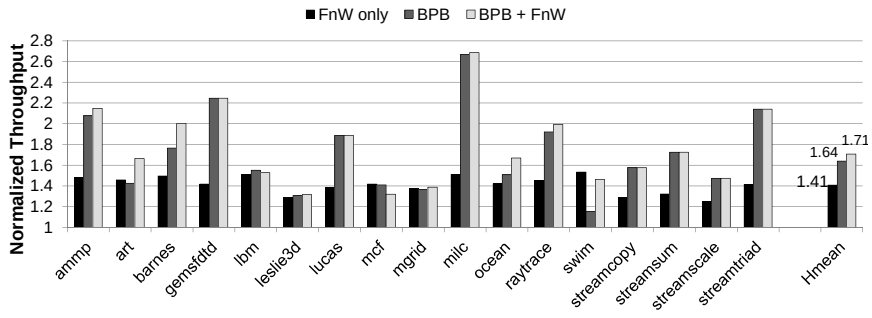
`FnW-only` reduces power demand by having an upper bound of bit changes, and may choose between 8-round and 4-round configurations. As a result, `FnW-only` improves throughput over baseline (simple power gating) by 32%∼41%. In general, both `BPB` and `BPB+FnW` bring forth additional throughput improvement (27%∼103%, 44%∼111%), because they can leverage fine-grained



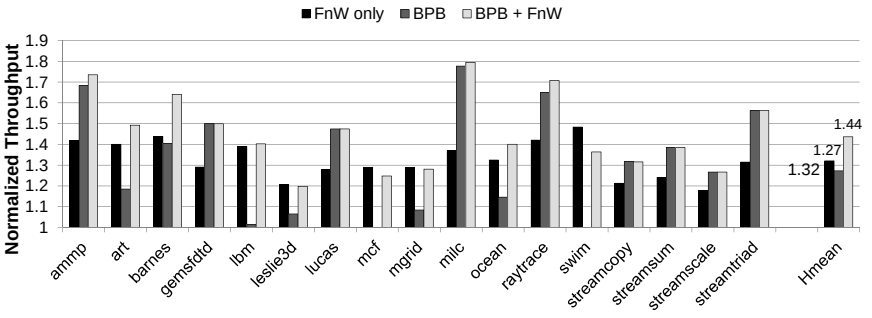
(a) 4×64 bit writes.



(b) 8×64 bit writes.



(c) 12×64 bit writes.



(d) 16×64 bit writes.

Figure 41: Comparing throughput improvement under different power budgets. RAWP is used as scheduler. Results are normalized to a baseline using RAWP with simple power gating.

bit change information and have more flexibility choosing write configurations. Of the two schemes, combining DW and FnW (`BPB+FnW`) is more effective because it further lowers number of bit changes (i.e., power demand) in each request.

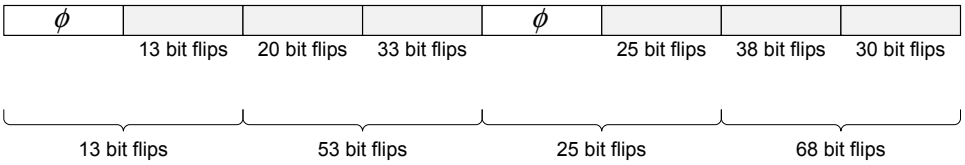


Figure 42: Example: DW with high number of bit changes.

BPB with DW alone (`BPB`) yields less average throughput improvement than `FnW-only` in one case (16×64 in Figure 40). The reason is that DW does not have an upper bound on bit changes. When dealing with requests that have a high number of bit changes, `BPB` could have higher power demand than `FnW-only`. Moreover, `BPB` is subject to the *DemandCap* limitation. These cause `BPB` to have less opportunity to choose configurations with fewer number of rounds. On the other hand, `FnW-only` has more chance of using the 4-round configuration when power budget is abundant. Moreover, BPB needs to pay extra overhead to compute $\hat{T}_{start}$ and pick the write configuration. These factors cause `BPB` to perform slightly worse than `FnW-only` under high power budget. Similar trends can also be seen workload-wise, as shown in Figure 41. I illustrate this case with an example. Consider a high bit change request as in Figure 42, and suppose available power budget is 64 bit writes. `BPB` would not choose 4-round configuration because its power demand for the 4-round configuration (68 bit writes) would exceed the available power budget. `FnW-only`, on the other hand, always assumes power demand of 64 bit writes for a 4-round configuration, so it can fit in the available power budget using a 4-round configuration. Although `BPB` can skip redundant rounds, it would still take longer time than `FnW-only`. Now if I lower the power budget to 40 bit writes, then both `BPB` and `FnW-only` can only choose the 8-round configuration. `BPB` is advantageous in this case as it can skip redundant rounds (which also explains why it performs better under lower power budgets).

By combining DW and FnW together in BPB, I effectively solved the above shortcoming and mitigated the problem (as shown in Figure 40 and Figure 41). In summary, my combined scheme (`BPB+FnW`) achieves 44% to 111% throughput improvement over a simple power gating scheme under different power budgets, and up to 70% more improvement than `FnW-only`.

An interesting observation from the experiments is that both `BPB` and `BPB+FnW` schemes become more and more effective when the power budget is decreasing. Note that the absolute numbers of throughput are decreasing as power budget gets lower, but the improvement I gain from BPB over the baseline increases. This is because, when available power budget is low, the baseline's parallelism is more severely limited as it assumes each round changes every bit. On the other hand, `BPB` and `BPB+FnW` leverage the fine-grained bit change information, so they can skip redundant rounds and can utilize the limited power budget more efficiently. This leads to more concurrent write requests at runtime and better throughput.

6.7 REMARKS ON THROUGHPUT IMPROVEMENTS

The throughput improvements achieved by RAWP in Figure 30(b) (Section 5.8.2) are based on an experiment in which power budgeting is disabled. Since the experiment uses a fixed 8-round configuration, the maximum number of concurrent bit writes that could be generated in this experiment is 16×64. If the power budget is set to 16×64 or more, the results in Figure 30(b) are same as using RAWP with a simple power gating scheme, which is the baseline in Figure 41(d). Therefore, the combined throughput improvements using RAWP and BPB+FnW over baseline blocking bank design (under the power budget of 16×64 concurrent bit writes) will be the multiplication of the improvements in Figure 30(b) and Figure 41(d), as shown in Figure 43.

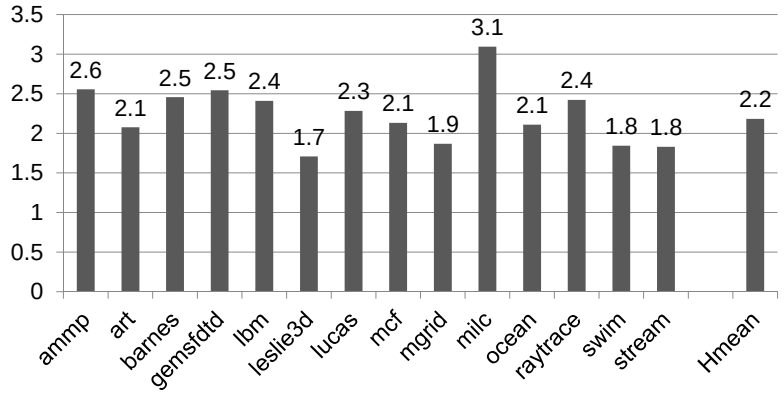


Figure 43: Combined throughput improvements over blocking bank design using RAWP and BPB+FnW. Power budget is 16×64 concurrent bit writes.

Another interesting question is: Instead of improving the throughput of PCM memory, can we just add more cache to achieve the equivalent performance? To answer this question, I use an analytical model from Moguls [91] to estimate the cache size increase if no throughput improvement technique is applied. The model approximates the correlation between cache size and its bandwidth requirement to next level. Essentially, a larger cache will have lower miss rate and will require lower throughput to the next level. To achieve the equivalent performance of a system with higher memory throughput, a system with lower memory throughput will need larger cache to reduce the bandwidth requirement to the memory. Figure 44 illustrates the estimated cache size increase if no throughput improvement technique is applied, compared to the case when RAWP and BPB+FnW are used. On average, we need a 4.8× larger cache if we do not improve the throughput of PCM memory. Note that I do not include streaming workloads in the figure, because increasing cache size does not help in these cases. These results help explain why improving throughput of PCM memory is necessary.

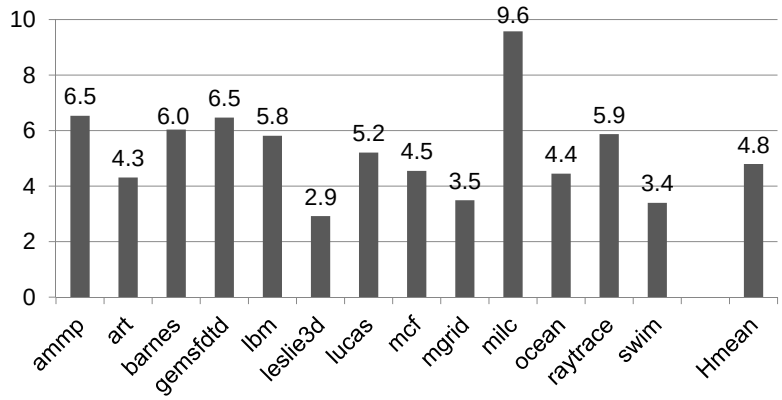


Figure 44: Estimated cache size increase if no throughput improvement technique is used.

7.0 IMPROVING QOS TUNING ABILITY

The techniques I proposed in previous two chapters aim at a lower level scheduling and are not aware of application priorities. In this chapter, I propose my techniques to improve QoS tuning ability for high-priority applications in PCM memory [102] (Figure 45).

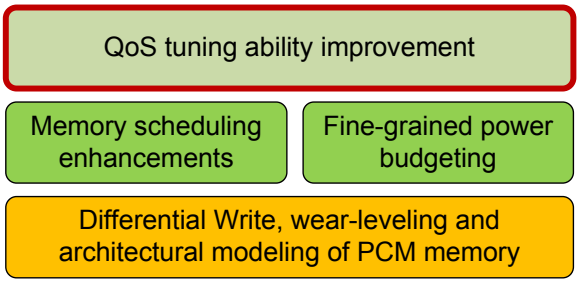


Figure 45: Overview of my research – QoS tuning ability improvement.

When multiple applications are running concurrently, their memory requests can interfere with each other and cause longer read latencies. For high-priority applications, it is often desirable by operating system or administrator to have the ability of controlling this "memory slowdown". However, PCM's long write latency worsens the interference and degrades this tuning ability. As I will show in this chapter, a high-priority application can still suffer from significant read latency increase when running concurrently with other applications, even its requests are given highest priority. Hence the tunable range of its read latency increase is limited, indicating poor QoS tuning ability. In this chapter I propose two techniques, Request Preemption and Row Buffer Partitioning, to mitigate this issue and extend the tunable range of high-priority applications' read latency increase.

7.1 QOS TUNING ABILITY ISSUE IN PCM MEMORY

In modern chip multiprocessor (CMP) systems, main memory is a critical shared resource for serving multiple applications running concurrently. The applications, ranging from high performance scientific computations, to streaming video processing, to data intensive data mining, often have large memory capacity requirements but different runtime memory access behaviors. The memory footprint of modern applications is usually large (e.g., a database application) requires large main memory to hold the index file for efficient query processing; and a web application requires large main memory to cache its recent accessed files. However the memory access behaviors are quite different at runtime. For example, a computation intensive application often has bursty memory accesses and low overall memory bandwidth requirement. Its performance is usually sensitive to the average processing latency of its memory accesses. In contrast, a memory intensive application has continuous accesses during the execution. Its performance is more sensitive to its acquired memory bandwidth at runtime.

When multiple applications are running concurrently, each application suffers longer memory access time and performance loss caused by interferences among the concurrently running applications. For some computation-intensive (memory non-intensive) applications, it is often desirable for the operating system or administrator to have some control over this kind of "memory slowdown". In other words, operating system or administrator should be able to tune the access latency increase of these computation-intensive applications. I term the range of tunable memory slowdown the *QoS Tuning Range* of these applications. A larger tuning range provides better QoS tuning ability, giving operating system or administrator better control over memory slowdown. Intuitively, tuning memory slowdown can be achieved by applying priorities to the memory accesses from different applications.

However, PCM's long write latency can greatly degrade QoS tuning ability when scheduling a mix of applications of different priority levels. For example, read requests from a high-priority application may arrive at the memory controller but find the bank is busy servicing requests from other applications. Due to PCM's long write latency, the high-priority application may still suffer from significant memory slowdown even though each of its requests is granted high priority by scheduler. Figure 46 illustrates the average read latency of computation-intensive applications (which are given high priority) when running concurrently with other applications (details of the mixes are explained in Section 7.4). The numbers are normalized to their average read latency when

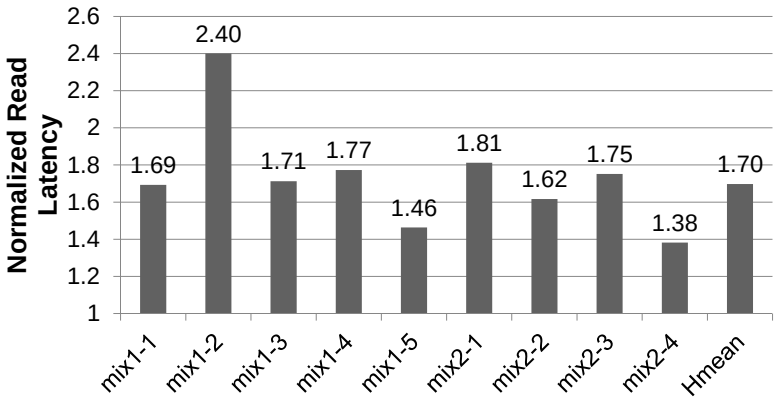


Figure 46: Average read latency of high-priority applications when running concurrently with other applications, results are normalized to their read latency when running alone.

running alone. I choose average read latency as it is a good indicator of the performance impact from the memory system – for computation-intensive applications, their write operations are not on the critical path due to the DRAM buffer before PCM memory. Each application mix has four applications while one or two applications are assigned to be of the highest priority. As we can see from the figure, the high-priority applications still suffers significant memory slowdown (70% on average) even though their requests are granted high priority in memory scheduling. Existing memory scheduling schemes [62, 63, 65] focus on fairness and overall throughput when scheduling requests from multiple applications. They were designed for DRAM systems that have different physical characteristics, and lack the ability to tune the scheduling to meet the QoS requirement in a PCM memory system.

7.2 ARCHITECTURE OVERVIEW

I use a similar 4-core CMP architecture as in previous chapters. PCM memory is used along with a DRAM buffer to mitigate its write latency and endurance. I assume 8 logic banks in PCM memory (meaning that it can serve 8 requests concurrently), each with a multi-entry row buffer. The multi-entry row buffer in my design is organized as a fully associative cache managed by LRU policy. Due to the same reason as explained in Section 5.1, I employ a write-through policy in this

row buffer design. Figure 47 shows an overview of my architecture. While the data of multi-entry row buffer may reside on either PCM DIMM or the memory controller, its tag array should reside on the memory controller for scheduling purposes.

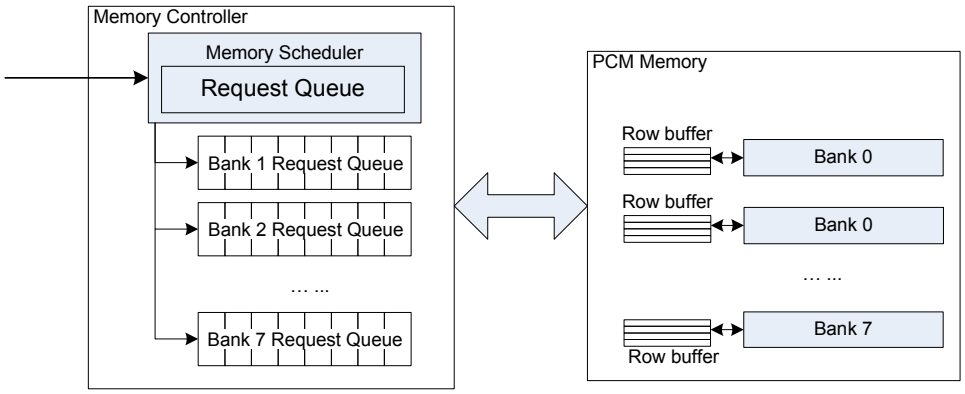


Figure 47: Baseline architecture used in my study.

The procedure of serving memory request is similar to previous chapters as well: Memory requests arriving at the memory controller are first buffered in the request queue. The memory scheduler then dispatches the requests to bank queues according to certain policy (e.g., PAR-BS [63]). Requests are then issued from the bank queues to corresponding logic banks.

I use *parallelism-aware batch scheduling* (PAR-BS) [63] as my baseline scheduler. PAR-BS groups a number of memory requests into a batch and ensures that all requests from the current batch are serviced before the next batch is formed. When forming a batch, PAR-BS marks up to `Marking-Cap` requests per each bank to promote fairness and bank-level parallelism. Requests within a same batch are ordering according to their ranks (details of PAR-BS are discussed in Chapter 2).

7.3 FINE-GRAINED QOS TUNING FOR PCM MEMORY

7.3.1 Problem Analysis

The PAR-BS used in my baseline has simple priority scheduling tuning ability. When scheduling a mix of applications with different priority levels, PAR-BS enhances the scheduling by allowing requests from high-priority applications to be dispatched before other requests in the batch.

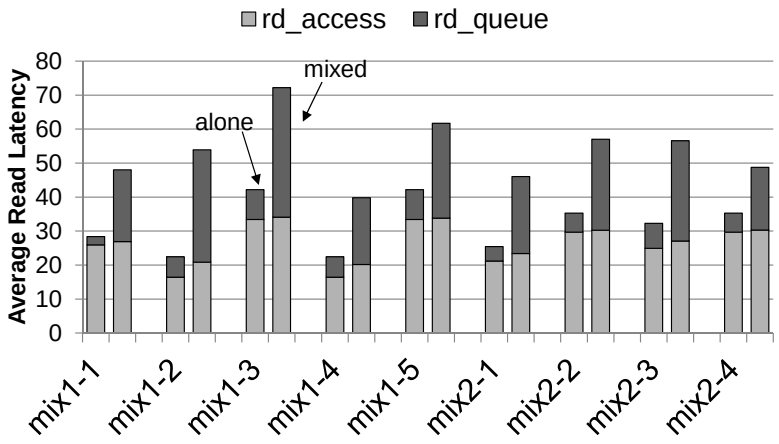


Figure 48: Breakdown of high-priority applications' average read latency when running concurrently with other applications.

Unfortunately, PCM's long write latency greatly degrades this tuning ability. Requests from high-priority applications still suffers from significant latency increase as I have shown in Figure 46. I analyzed the results and break down the increase of average read latency into two parts, access latency and queuing latency, as shown in Figure 48. The access latency is the time to serve the actual request, and the queuing latency is the time request spent in the queues. As we can see, the increase of average read latency comes from two sources:

- **Increased queuing time.** When running concurrently with other applications, requests from high-priority applications spend more time in the queues to wait for previous requests to finish. And since PCM write is slow, this increases queuing latency significantly (4.72× on average).
- **Increased row buffer misses.** When multiple applications are running concurrently, requests from multiple applications compete for the small multi-entry row buffer and cause more row buffer miss to high-priority applications. This accounts for 6% increase in average read latency.

This significant memory slowdown of high-priority applications limits QoS tuning range, and degrades QoS tuning ability that operating system or administrator could have. In order to address this challenge, I need to reduce latency increase from both sources. In the following sections I propose two techniques, Request Preemption and Row Buffer Partitioning, to reduce queuing time and improve row buffer hit rates for high-priority applications.

7.3.2 Request Preemption

Based on the above observation, my first design targets at reducing the queuing time of high-priority requests. Since the priority-based batch scheduling still needs to complete the current batch before the next batch is formed, a high-priority request, when it arrives at the memory controller, may experience a long queuing time. If this request can preempt the current being serviced requests, then the queuing time could be greatly reduced. I only allow read requests to preempt other requests as they are the requests on the critical path.

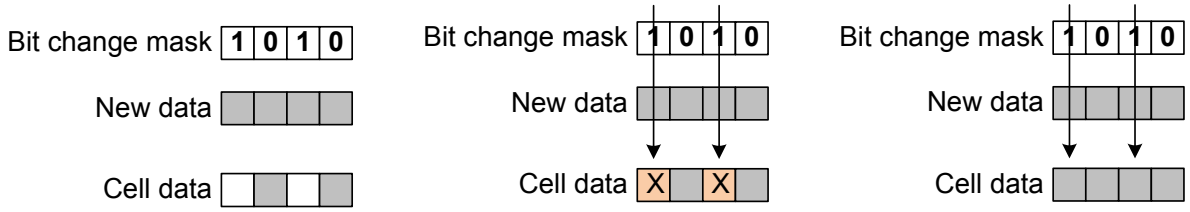
Batch preemption. A simple preemption is to insert incoming high-priority requests into the current batch (i.e., the current batch can dynamically accept grow and mark more requests). Within the batch, the incoming requests are scheduled behind those with the same or higher priority levels. If no such request exists, then the incoming requests are serviced right after the completion of the current request.

Batch preemption may starve low priority requests if they are blindly scheduled after the high-priority requests, and there may be continuous incoming high-priority requests. To avoid this side effect, my batch preemption adopt a similar idea as the "Empty-Slot Batching" in PAR-BS [63]. For high-priority threads, I allow their requests to be "inserted" into the *current* batch even they arrive after the batch is formed, as long as they do not exceed their `Marking-Cap` limitation (in PAR-BS, each thread can have up to `Marking-Cap` requests per bank in a batch). Note that the issued requests in current batch are not regarded as empty slots, hence high priority threads will not be able to insert requests indefinitely as they will eventually use up their `Marking-Cap` quota.

Access preemption. Even with batch preemption, high-priority requests still need to wait for the completion of the current PCM access even if the batch has no other high priority request. Since PCM write accesses are much slower than DRAM, the effect of using batch preemption alone is limited – I still observe significant performance loss in high-priority applications.

To further reduce the request queuing time of high-priority applications, I exploit the non-volatility of PCM cells and allow a high priority request preempting the current low priority PCM access. That is, the memory controller sends a *preempt* command to stop the ongoing PCM access, and then an *activate/write* command starts a new one. As a comparison, it is impossible to preempt a DRAM access due to its destructive read — DRAM read destroys the contents of a whole row. Moreover, preempting requests in DRAM does not have much motivation as DRAM's read and write have symmetric speed.

The current access can be one of row buffer hit, row buffer read miss (i.e., PCM read), or PCM write operations. The memory controller never preempts a row buffer hit due to its low latency. Since PCM reads are non-destructive, preempting a PCM read leaves no change to the PCM cells. The preempted read can be restarted at a later time. Preempting a PCM write is a bit complicated. With the incorporation of *Differential Write* technique [103], a write operation includes a pre-write read of the cells for comparison. The cells are unchanged if preemption happens before real write operation starts, and left in undetermined states if preemption happens when the cells are being modified. In my current design, I adopted the pessimistic assumption — the cell states are undetermined, and the memory controller needs to resend the line data to restart.



(a) With Differential Write, bit change masks are generated.

(b) PCM write preempted, bit change masks and new data are saved.

(c) PCM write is restarted, only the cells that might be in undetermined state are rewritten.

Figure 49: Preemption of a PCM write request.

When a PCM write is preempted, the cells that are being written may be left in undetermined state. Hence its new data and bit change mask (generated by Differential Write) are saved for the request to restart later. The saved bit change mask allows us to write only the cells that are previously written (and may be in undetermined state) when the write request is restarted, instead of writing the whole line. This procedure is illustrated in Figure 49. In other words, I inherit the benefits of Differential Write in my technique.

While it is safe to preempt a PCM read or write at any time, it is not always a better decision, in particular, preempting a PCM access that is close to finish introduces both performance and energy overheads. Another drawback is, a write operation may be preempted and restarted several times before its completion, which reduces the write endurance of affected cells. In my design, I set a threshold `T` and disable preemption if the current access can finish in `T` cycles. A preemption threshold helps to tune the trade-off between overheads and performance impacts to the high-priority applications.

7.3.3 Row Buffer Partitioning

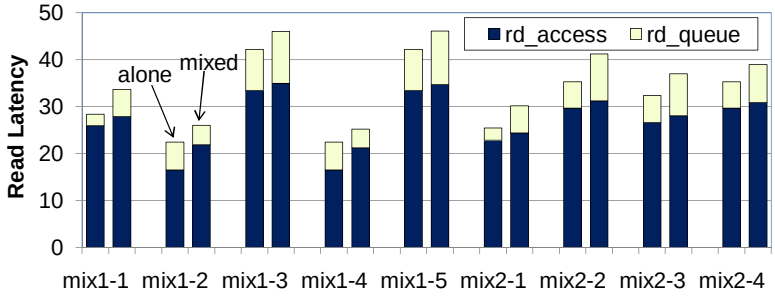


Figure 50: Read latency breakdown of high-priority applications with request preemption enabled.

While preemption helps to reduce the long queuing time of high-priority applications, the increased contention of requests from multiple applications destroys the row buffer locality and results in more row buffer misses. Figure 50 illustrates the read latency breakdown of high-priority applications when Request Preemption is enabled. We can see that Request Preemption effectively reduce the queuing time increase when multiple applications are running concurrently. However, the reordering of high-priority requests increases row buffer miss, resulting in more access time increase (9%) for high-priority applications. The increase was 6% when using the priority-based batch scheduling. With the reduction of queuing time, the increased access time contributes more portion in memory slowdown of high-priority applications. Hence a technique to control the row buffer miss rate is needed as well.

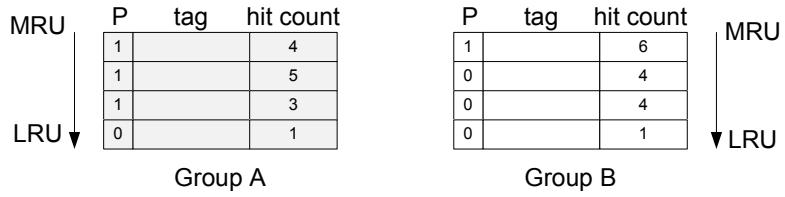
To provide fine-grained QoS tuning ability, I propose to control row buffer misses by dynamically partitioning the row buffer entries among concurrent applications. My design is motivated by the utility based partitioning for caches [78].

7.3.3.1 Utility-based partitioning. Suppose a cache (or multi-entry row buffer in my case) is partitioned between two groups (Group A and Group B). Utility-based cache partitioning [78] (UCP) scheme tries to answer the question: If I grab one entry from group A and give it to group B, how much gain and loss will I get?

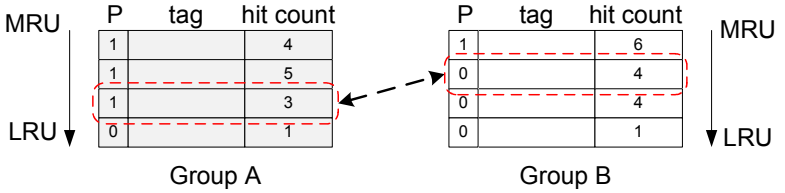
In order to answer this question, UCP maintains two tag arrays, one per each group. Both groups manage their tag array using LRU policy. In addition, each tag has a hit counter. The hit counters count the number of hits at each position for each group at runtime, and thus provide the utility information when different number of entries are allocated to each group. Since data array

is not doubled (meaning that each data entry either belongs to group A or group B), a "P" bit is added to each tag to indicate whether actual data presents for this tag (i.e., whether there is a data entry allocated for this tag). Apparently, "P" bits represent the current partitioning of the cache, and the number of P=1 tags in both tag arrays should equal to the size of the cache set. Figure 51(a) shows an example of partitioning in a cache with 4 entries. As shown in the figure, each group has its tag array and hit counters.

By having two tag arrays with hit counters, UCP can track the utility information from both groups. Due to the *inclusive property* of LRU replacement policy, that is, an access hit in a 2-entry cache is also a hit in a 3-entry cache, UCP can evaluate the gain and loss by comparing the hit counters from both groups. As shown in Figure 51(b), it answers the above question by comparing the two hit counters from group A and group B: 1) the hit counter of group A's LRU entry whose P bit is 1; 2) the hit counter of group B's MRU entry whose P bit is 0. In this example, removing one entry from group A would lose 3 hits, but giving it to group B would increase hits by 4. So UCP would predict that this repartitioning is beneficial.



(a) Current partition.



(b) Evaluating gain and loss.

Figure 51: Utility-based cache partitioning between group A and B.

7.3.3.2 Row buffer partitioning in my design. I adopted similar idea as UCP in my row buffer partitioning scheme. I divide the applications into two groups: the prioritized group (PG) that contains high-priority applications; and the normal group (NG) that contains all other applications. Each row buffer entry belongs to one group at a time while the number of entries allocated

to each group is dynamically determined by evaluating their utility of the entries. In order to support the tuning of PG's row buffer miss, I assign *weights* to both groups and use the weights when evaluating the gain and loss on repartitioning.

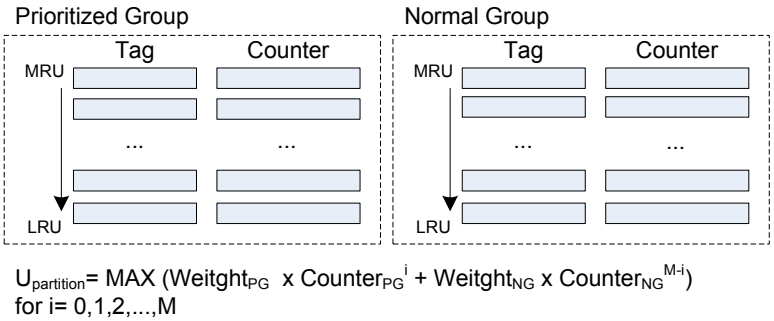


Figure 52: Partitioning the row buffer entries between normal group and priority group based on their utility and weights.

Figure 52 illustrates the monitoring logic in my design. For an M-entry row buffer, each group maintains an M-entry tag array and an M-entry counter array. Both are ordered according to their recency positions, i.e., from MRU (most recent used) to LRU (least recent used). For the example in Figure 52, there are M+1 possible cases: one group gets 0,1,2, ..., and M entries respectively while the other group has all the rest entries. I divide the execution into epochs and periodically adjust the partition for the next epoch based on the utility of the past epoch. To simplify the computation at runtime, each repartitioning involves one step (i.e., grab one entry from a group to another each time).

7.4 EVALUATION

7.4.1 Experimental Settings

I evaluated my proposed scheme through trace-driven and execution-driven simulations. Traces are collected from a Simics-based simulator [56] and processed to resemble the effects of load-store queue and write buffer. I used an in-house trace-driven simulator to evaluate the MCPI (memory cycles per instruction), average read latency and unfairness indexes of different schemes. A GEMS/Simics-based execution-driven simulator [57] was then used to evaluate the average IPC

(instruction per cycle). The settings are summarized in Table 7. I assumed a similar 3D stacked PCM memory as in Chapter 4, but my techniques can be applied to off-chip PCM memory as well. The PCM write latency is conservative to my results, as longer PCM write latency could worsen the interference and result in more significant improvements.

Table 7: Experimental platform used in my evaluation.

Components	Parameters
Processor Core	4 cores, each core is 4-issue, out-of-order, run at 1GHz
L1 Cache	32K I-cache and 32K D-cache, 64B cacheline size, 4-way set associative, 3-cycle cache hit latency
L2 Cache	512KB per core, 64B cacheline size, 16-way set associative, 6-cycle hit latency
Main Memory	4GB PCM memory, 8 logic banks read latency: 10ns (row hit) or 30ns (row miss), write latency: 100ns

I assembled a set of applications with different memory bandwidth requirements from SPEC SPEC2006 [17], SPLASH2 [88], and STREAM [58] benchmark suites. For each application, I skipped its warm-up phase, and simulated 100 million instructions for memory access and IPC studies. Table 8 lists the characteristics of the simulated phase for each application. They are categorized into two groups based on MCPI (memory cycles per instruction): the computation-intensive applications that include bzip2, fmm, and raytrace, and the memory intensive applications that include others. Memory intensive applications have much higher MCPI than memory non-intensive ones.

I then mixed these applications to construct a set of multiprogramming workloads to evaluate scheduling effectiveness. As shown in Table 9, each mix contains 4 applications. Applications are picked based on their memory-intensiveness (MCPI). Three groups of mixes are formed. For each `mix1-*` workload, there is one high-priority application while for each `mix2-*` mix, there are two high-priority applications. The high-priority applications are in bold font in the figure. While high-priority programs are usually computation-intensive (memory non-intensive) in practical, I also evaluated the cases in which high-priority applications are also memory-intensive (`mix3-*`).

To evaluate the effectiveness of my proposed scheduling scheme, I implemented and compared the following three approaches in my experiments.

Table 8: Benchmark characteristics.

Benchmark	MCPI	Type	Comment
bzip2	0.085	Int	SPEC2006, non memory-intensive
fmm	0.297	FP	SPLASH2, non memory-intensive
raytrace	0.044	Int	SPLASH2, non memory-intensive
gemsfdtd	1.190	FP	SPEC2006, memory-intensive
lbm	3.618	FP	SPEC2006, memory-intensive
leslie3d	1.281	FP	SPEC2006, memory-intensive
mcf	3.927	Int	SPEC2006, memory-intensive
milc	9.816	FP	SPEC2006, memory-intensive
stream	2.534	Int	STREAM, memory-intensive

Table 9: Workload mixes.

Mix	Applications (high-priority ones are in bold font)
mix1-1	mcf, milc, stream, **raytrace**
mix1-2	mcf, milc, stream, **bzip2**
mix1-3	mcf, milc, stream, **fmm**
mix1-4	lbm, leslie3d, gemsfdtd, **bzip2**
mix1-5	lbm, leslie3d, gemsfdtd, **fmm**
mix2-1	milc, lbm, **raytrace, bzip2**
mix2-2	lbm, milc, **fmm, raytrace**
mix2-3	lbm, milc, **fmm, bzip2**
mix2-4	gemsfdtd, milc, **fmm, raytrace**
mix3-1	mcf, stream, gemsfdtd, **leslie3d**
mix3-2	mcf, milc, stream, **gemsfdtd**

- `PAR-BS (baseline)`. This implements the PAR-BS scheduling scheme [63] without priority levels (i.e., all applications have same priority, no QoS tuning enhancements).
- `PAR-BS/P`. This implements the PAR-BS scheduling with priority levels. I assign the highest priority level to the high-priority application(s) such that within each batch, the requests from high-priority programs are serviced before those from other applications.
- `BatchP`. This scheduling scheme enables Batch Preemption. It allows the insertion of high-priority requests into the current batch as long as its thread does not exceed `Marking-Cap` limit (similar to the Empty-Slot Batching idea in [63]).
- `FullP`. This scheduling scheme implements Batch Preemption, Access Preemption, and row-buffer partitioning. I restricted the number of times a request can be preempted to avoid excessive impact on low-priority applications.

I focused on evaluating the read access latency as it determines the memory impact on application performance. With the DRAM buffer between the last level cache and the PCM memory, the write operations are usually less critical for computation-intensive applications. Similar to the techniques I proposed in previous chapters, I used Weighted Speedup [63, 89] as the performance metrics.

7.4.2 Parameters

There are two important parameters in my scheme that can be used to tune the slowdown of high-priority applications. The first parameter is `preemption threshold T`. In Access Preemption, current PCM access cannot be preempted if it will finish in T cycles. Using a larger T value results in less preemption while using a smaller T results in more aggressive preemption. In particular, if T = 0, then the high-priority read can always preempt; if T equals write latency, then there is no preemption at all. Note that instead of using "remaining percentage", I use "remaining cycles" for more accurate control. The reason is that PCM read and write have different latencies, and therefore same remaining percentage means different remaining cycles for read and write.

The second parameter is `row buffer partition weight ratio W`. This is the weight ratio between prioritized group (PG) and normal group (NG), that is:

$$W = \frac{Weight\ of\ the\ Priority\ Group}{Weight\ of\ the\ Normal\ Group}.$$

By using these parameters, I can control the slowdown of high-priority application in fine granularity (details of analysis are presented in Section 7.4.6 and Section 7.4.7). I also use these analysis to decide the optimal settings (T=10, W=4 for mix1-* and W=2 for mix2-*) which are used in rest of my experiments.

7.4.3 Tunable QoS Range

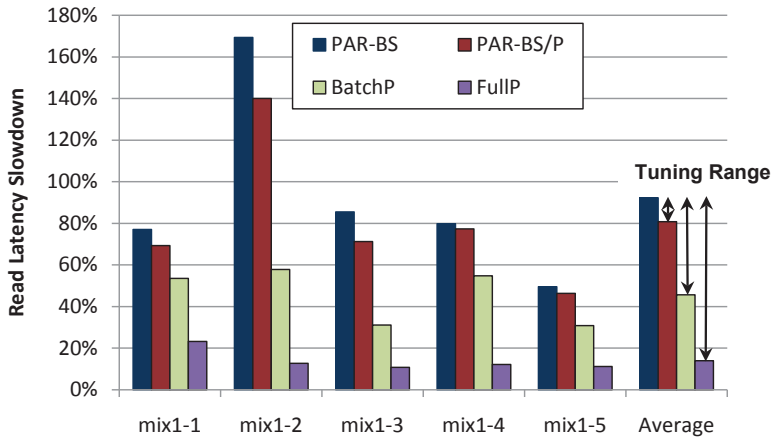
I first studied the tunable QoS ranges when using different memory scheduling enhancements. Since memory read is more critical to application's performance, I use *increase of read access latency (read slowdown)* as the metrics for tunable QoS range. In other words, my goal is to achieve a broader tunable range of high-priority application's read latency increase. In Figure 53(a), I set the baseline by using the PAR-BS without priority, and observed an average 92% increase of read latency for the high-priority application. When the high-priority application is assigned with the highest priority, `PAR-BS/P` reduces its slowdown to 81% on average. That is, `PAR-BS/P` has a tunable QoS range [81%, 92%]. `BatchP`, and `FullP` reduce the average slowdown to 46% and 14%, and achieve tunable QoS ranges [46%, 92%] and [14%, 92%] respectively. The tunable QoS range of `FullP` is 7× and 1.7× that of `PAR-BS/P` and `BatchP` respectively.

When there are two high-priority applications, `PAR-BS/P`, `BatchP`, and `FullP` have tunable QoS ranges [67%,72%], [48%,72%], and [20%,72%] respectively. `FullP`'s range is 10× and 2.2× that of `PAR-BS/P` and `BatchP` respectively. Using `FullP`, the high-priority applications still have performance losses comparing to their stand-alone execution. The reason is that there are non-removed interferences from other applications (e.g., a close-to-finish PCM request cannot be preempted due to energy and overall performance efficiency considerations).

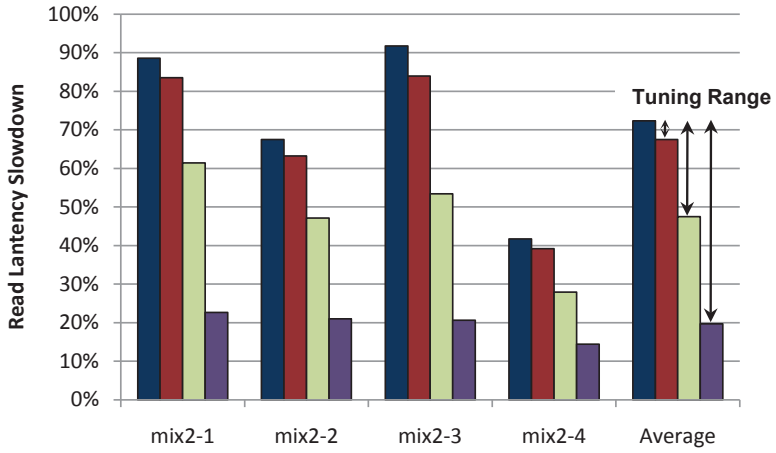
In summary, by providing a wider QoS tunable range, my scheme enables better system control for the high-priority programs running in multiprogramming CMP environment.

7.4.4 Fairness and Throughput

Since the main memory is shared, improving the performance of high-priority applications tends to slowdown other applications. Figure 54 presents the average read latencies for high-priority and low-priority applications respectively. My scheme successfully reduces the read latency of high-priority applications without incurring large overhead to other applications. For example, when there is one high-priority application, the average read latency of the high-priority application

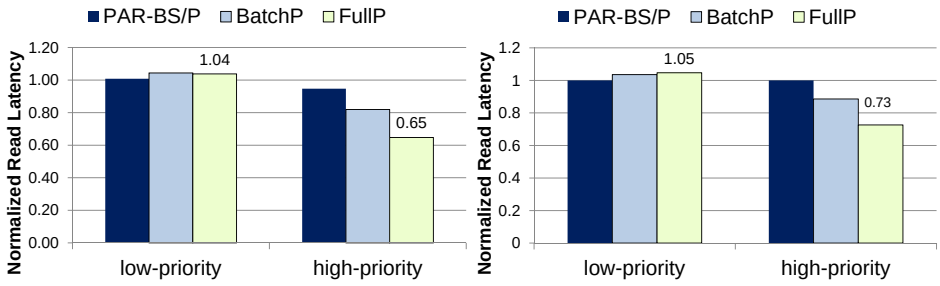


(a) Mix1-* workloads.



(b) Mix2-* workloads.

Figure 53: Tunable ranges of read latency increase (for high-priority applications).



(a) Mix1-* workloads.

(b) Mix2-* workloads.

Figure 54: Average read latency of high-priority and low-priority applications.

reduces 35% while that of the high-priority applications increases around 4%. This is because the high-priority applications are not memory intensive in the mixes. My scheduling scheme did not allocate more bandwidth than its preassigned share.

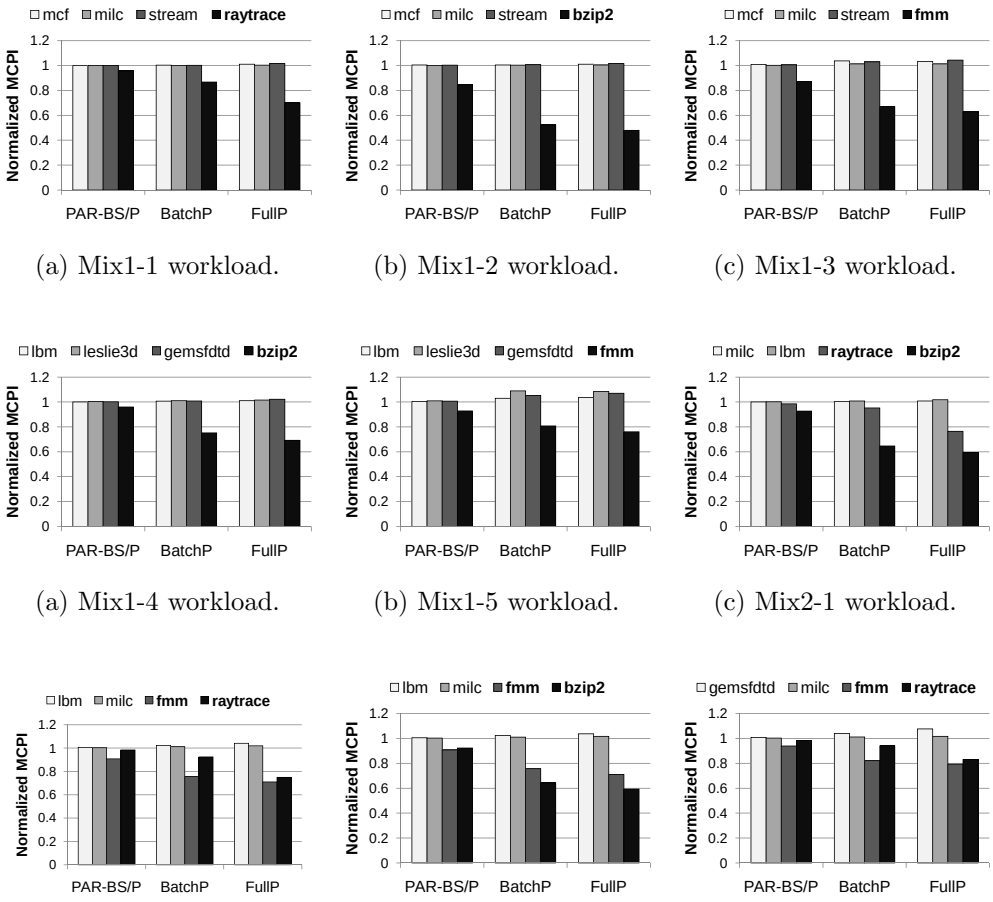
Figure 55, Figure 56 and compare the MCPI (memory cycle per instruction), weighted throughput and IPC of individual applications in selected mixes. I normalized the MCPI to PAR-BS without priority. My scheme has small impacts on MCPI of low-priority applications. The loss is within 2% for mix1-* and mix2-* workloads. The changes to the overall throughputs are also small.

As expected, `FullP` has a relatively large impact on fairness (shown in Figure 57). For example, my fairness is worse than `PAR-BS/P` due to large performance improvement of the high-priority application. When there are two applications, there is small fairness improvement since the two high-priority applications amortize the improvement from each other.

7.4.5 Prioritizing Memory-Intensive Applications

While high-priority programs are typically memory non-intensive (i.e., computation-intensive), I also evaluated the cases in which high-priority applications are also memory-intensive for completeness (Figure 58). Due to increased number of memory requests, prioritizing memory-intensive application results in more significant impacts on low-priority applications. For example, MCPI of low-priority application in mix3-1 is increased by 13% on average.

Another note is that from `BatchP` to `FullP`, high-priority application's MCPI does not reduce as much as its average read latency. Taking mix3-2 as example, high-priority application's MCPI

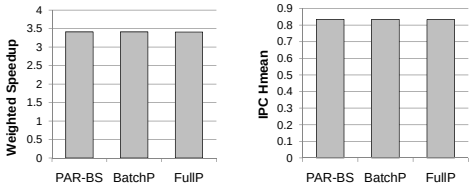


(a) Mix1-1 workload. (b) Mix1-2 workload. (c) Mix1-3 workload.

(a) Mix1-4 workload. (b) Mix1-5 workload. (c) Mix2-1 workload.

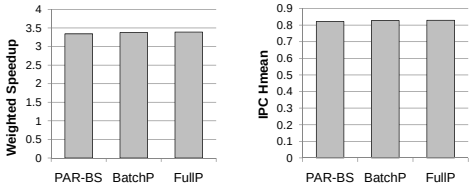
(d) Mix2-2 workload. (e) Mix2-3 workload. (f) Mix2-4 workload.

Figure 55: Comparing normalized MCPI using different scheduling enhancements.



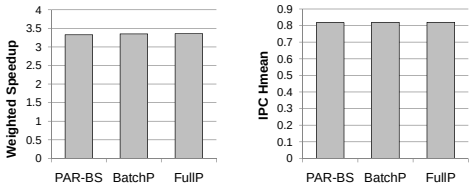
Weighted speedup Average IPC

(a) Mix1-1 workload.



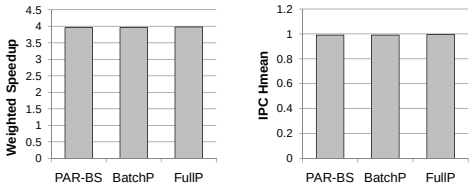
Weighted speedup Average IPC

(b) Mix1-2 workload.



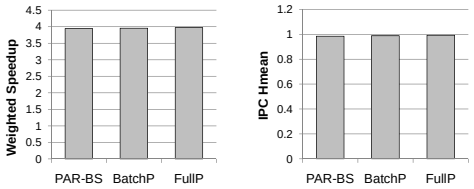
Weighted speedup Average IPC

(a) Mix1-3 workload.



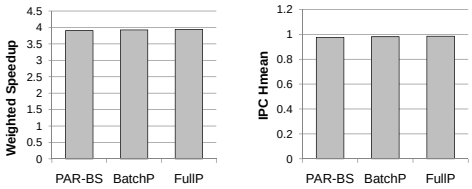
Weighted speedup Average IPC

(b) Mix1-4 workload.



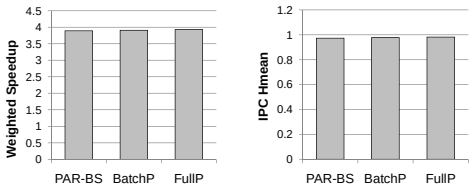
Weighted speedup Average IPC

(c) Mix1-5 workload.



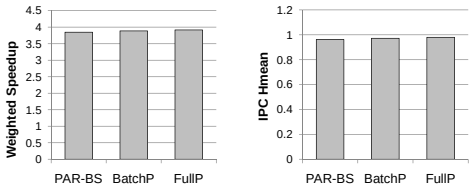
Weighted speedup Average IPC

(d) Mix2-1 workload.



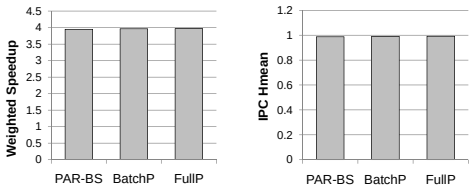
Weighted speedup Average IPC

(e) Mix2-2 workload.



Weighted speedup Average IPC

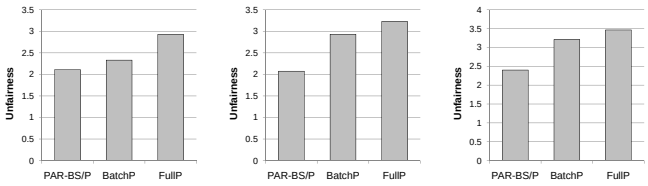
(f) Mix2-3 workload.



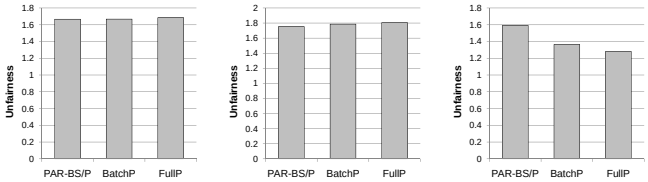
Weighted speedup Average IPC

(e) Mix2-4 workload.

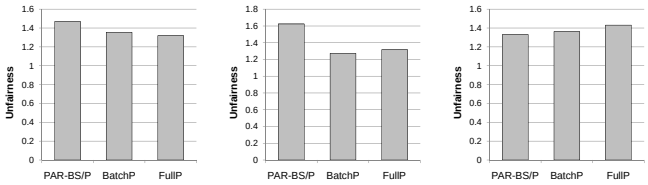
Figure 56: Comparing weighted throughput and IPC using different scheduling enhancements.



(a) Mix1-1. (b) Mix1-2. (c) Mix1-3.

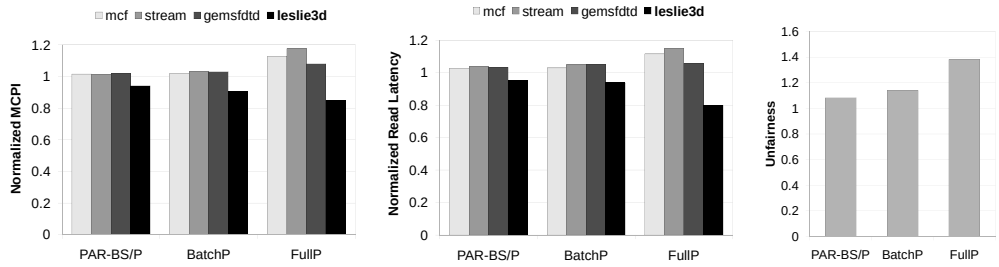


(a) Mix1-4. (b) Mix1-5. (c) Mix2-1.



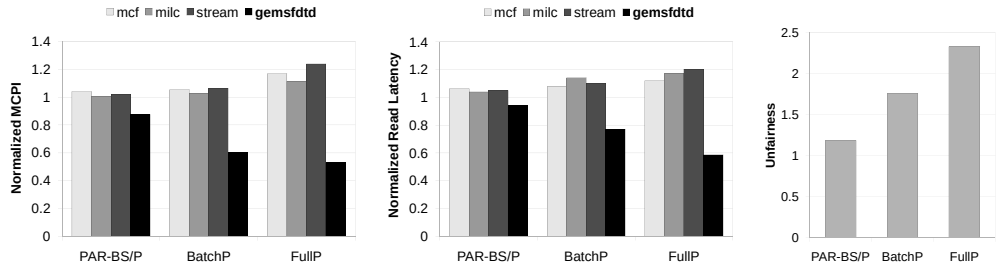
(d) Mix2-2. (e) Mix2-3. (f) Mix2-4.

Figure 57: Comparing unfairness using different scheduling enhancements.



MCPI Average read latency Unfairness

(a) Mix3-1 workload.



MCPI Average read latency Unfairness

(b) Mix3-2 workload.

Figure 58: Comparing MCPI, read latency and unfairness when high-priority applications are also memory-intensive.

is reduced from 0.60 to 0.53, while its read latency is reduced from 0.77 to 0.58. The underlying reason is that in `FullP`, only read requests from high-priority application can do Access Preemption, meaning that write requests from high-priority application are still scheduled in the same way as they are in `BatchP`. Since memory-intensive applications tend to have more write requests (e.g., 42% of memory requests are writes in `gemsfdtd`), and PCM write takes much longer time than read, the MCPI reduction from `BatchP` to `FullP` is not as significant as average read latency.

7.4.6 Preempt Threshold

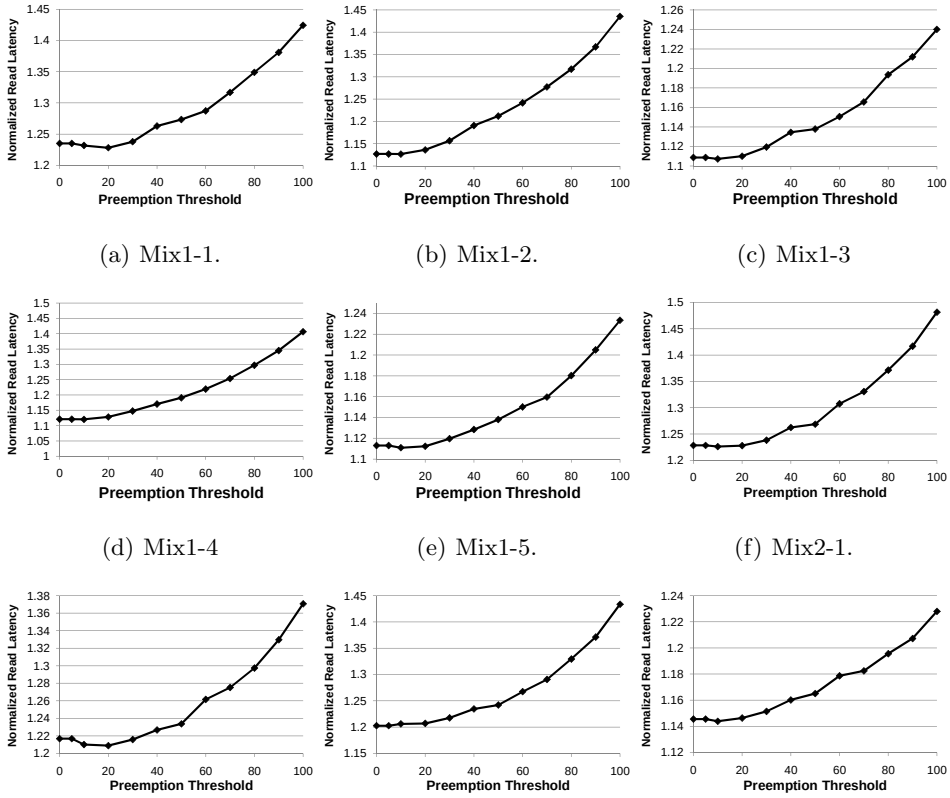
As described in Section 7.4.2, preemption threshold T is used to determine the aggressiveness of preemption. I assume one memory cycle penalty for each Access Preemption, so smaller preemption threshold does not always result in best average latency. Therefore, the trade-off needs to be studied to decide the optimal threshold value.

Figure 59 plots the normalized read latency of high-priority applications using different preemption thresholds. From my experiments, I achieved best trade-off when T = 10 cycles, which was the threshold I chose in other experiments. As a comparison, if T = 100 cycles (i.e., there is no preemption), the normalized read latency of the high-priority application in mix1-2 workload increases from 1.13 to 1.44.

7.4.7 Weight Sensitivity

The row buffer partition weight ratio W affects how row buffer is partitioned between Priority Group and Normal Group. Figure 60 shows the average read latencies of all applications and high-priority applications with different weight ratios. Figure 61 illustrates the normalized row buffer hit rates for the high-priority application when using different weight ratios.

When there is one high-priority application (i.e., for the workloads mix1-*), W=4 gives the best trade-off — the read latency averaged over all programs in each mix keeps low while the read latency of high-priority applications has a drop for 3 out of 5 mixes. The row buffer hit rates jump for 3 out of 5 mixes. When there are two high-priority applications (i.e., for the workloads mix2-*), W=2 and W=4 are potential candidates. W=2 weights more on the overall system performance, while W=4 weights more on high-priority applications. In my experiments, I chose W=4 for mix1-* and W=2 for mix2-* workloads respectively.

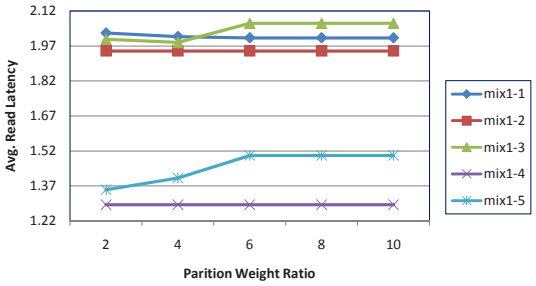


(a) Mix1-1. (b) Mix1-2. (c) Mix1-3

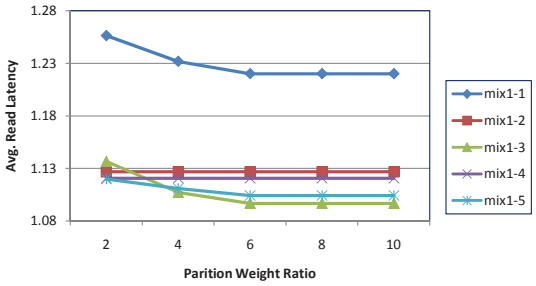
(d) Mix1-4 (e) Mix1-5. (f) Mix2-1.

(g) Mix2-2. (h) Mix2-3. (i) Mix2-4.

Figure 59: The normalized read latency changes for high-prority application(s) with different preemption thresholds.

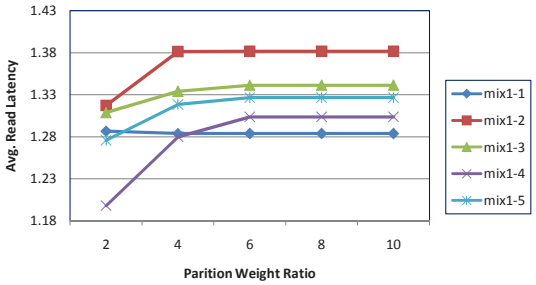


Average read latency (all applications).

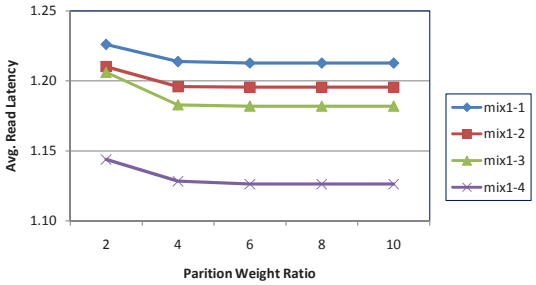


Average read latency (high-priority application).

(a) Mix1-* workloads.



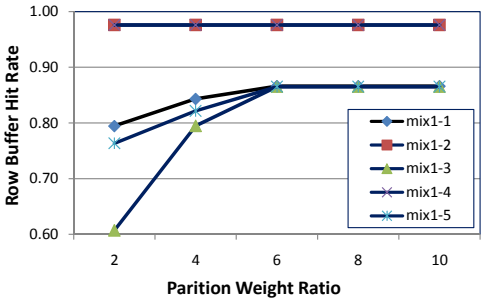
Average read latency (all applications).



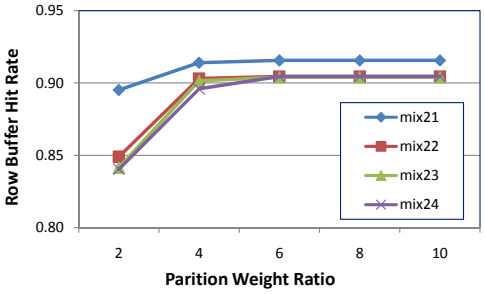
Average read latency (high-priority applications).

(b) Mix2-* workloads.

Figure 60: Normalized average read latency with different weight ratio W.



(a) Mix1-* workloads.



(b) Mix2-* workloads.

Figure 61: Normalized row buffer hit rates with different weight ratio W.

7.4.8 Number of Entries

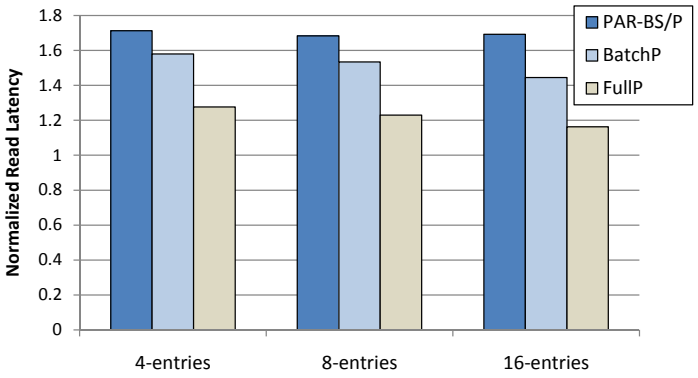


Figure 62: Comparing different number of row buffer entries.

Next I studied the impact when having different number of row buffer entries. I assume same row buffer storage budget in my study (i.e., keep the total storage of row buffer constant to 2KB). If the row buffer has 4 entries, then each row is 512B, while if using 16 rows, then each row is 64B. Figure 62 shows the normalized read latency of high-priority applications when using different settings of row buffer. I observed small improvements when more entries are used. In my other experiments, I used 8-entry row buffer.

7.4.9 Overhead Estimation

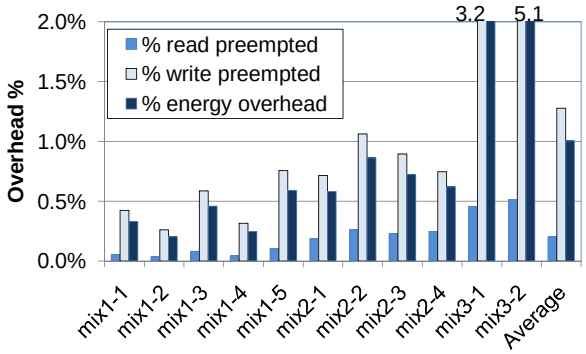


Figure 63: Preemption overhead estimation. Descriptions of mixes are summarized in Table 9.

Figure 63 plots read/write access and estimated energy overheads of my scheme. When a read or write operation was preempted, it has to be issued again and thus wastes the energy that it

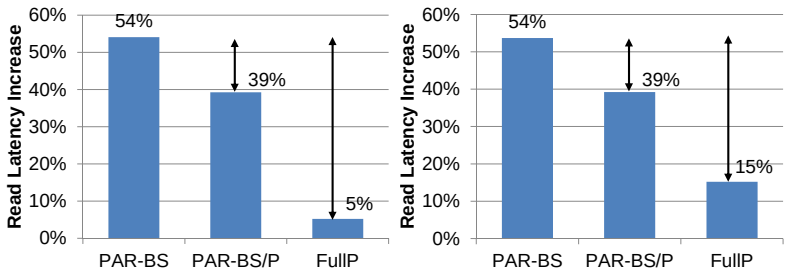
has consumed. From the figure, I preempted <1% read and write operations in most cases. This corresponds to about 1% energy increase on average. In this study I pessimistically assumed that a preempted PCM access wasted the energy to perform a full read or write.

The storage overhead in my scheme comes mainly from supporting row buffer partition. I used two priority groups, and added tag and counter array for each group. Since the PCM memory I used in experiments can support 8 outstanding memory requests (8 logic banks), the buffer size is 8× 2 groups × 8 entries/group × (3B/tag-entry + 4B/counter-entry) = 896B. Hence the storage overhead is negligible.

Similar to AWP and RAWP, my QoS improvement techniques can be implemented as additional subroutines in PAR-BS firmware. The computation overhead is relatively small compared to the PAR-BS algorithm.

7.5 REMARKS

The techniques I discussed in this chapter can improve the QoS tuning ability for PCM memory. An interesting question is: What if the techniques are applied to other memory technologies? For example, what if read requests and write requests are of the same speed? I estimate the QoS tuning ranges for two hypothetical cases: 1) slower read requests (read requests are as slow as write requests); 2) faster write requests (write requests are as fast as read requests). In the first case, I set the read access latency to be the same as original write access latency. And in the second case, I set the write access latency to be the same as original read access latency. My estimations of QoS tuning range are shown in Figure 64. `FullP` still achieves larger QoS tuning range than `PAR-BS/P` (3.3× and 2.7× larger respectively), although the gain is smaller due to the symmetric read and write speed in these hypothetical cases.



(a) Slower read requests. (b) Faster write requests.

Figure 64: Estimated QoS tuning range for two hypothetical cases.

8.0 EXPERIMENTAL INFRASTRUCTURE

In this chapter, I describe the experimental infrastructure I have built for my research as these experiences could be helpful to other researchers in this area.

8.1 SIMULATOR SETUP

8.1.1 Simics

The simulator I used is Simics [56] 3.0.31. Simics is an efficient full-system simulator which can simulate multiple instruction sets and platforms. The machine I simulated with Simics is Sun UltraSparc running Solaris 9. Here are my steps to setup my Simics environment:

1. Extract the Simics 3.0.13 package (tarball).
2. Create a workspace folder, and execute the Simics setup command in the folder:
   `<simics_folder>/bin/workspace-setup`
3. Copy the state files into the workspace folder: abisko-sol9.state, abisko-sol9-p1.state, abisko-sol9-p2.state.
4. Copy the Solaris 9 disk image (or create symbolic link) into the workspace folder.
5. Edit abisko-sol9.state, make sure that `checkpoint_path` is pointed to workspace folder, and `sd0_image` points to disk image file.
6. Create a Simics configuration file (.simics) under folder `<simics_folder>/targets/serengeti` to define a 4-core CMP system. The easiest way is to modify an existing configuration file.
7. Boot the simulated machine into Solaris:

   ```
   cd <workspace_folder>
   ./simics targets/serengeti/<simics_config_file>
   ```

8. After Solaris is started, I can interrupt the simulator using `<Ctrl-C>` at any time, and save current state (checkpoint) using command:

   ```
   write-configuration <checkpoint_file>
   ```

9. The saved checkpoint can be resumed using the "-c" option of simics command:

   ```
   ./simics -c <checkpoint_file>
   ```

Simics also allows setting breakpoints in programs. This is achieved by adding "magic instructions" into the source code and recompiling the code.

8.1.2 GEMS

The Multifacet General Execution-driven Multiprocessor Simulator (GEMS) [57] provides detailed models for cache, memory, out-of-order core and interconnects. GEMS is not a standalone simulator. Instead, it runs under Simics framework. (Simics also has a cache/memory model but it is rather simple.) GEMS consists of several modules: Ruby (cache/memory model), Opal (out-of-order core model), Garnet (interconnect model). My steps to setup GEMS under Simics is as follows.

1. Extract GEMS package, setup a Simics workspace under the GEMS folder.
2. Edit `<gems_folder>/scripts/makesymlinks.sh` and modify following line:

   ```
   ln -s <simics_folder>/import import
   ```

3. Run `makesymlinks.sh` under the workspace folder.

   ```
   cd <gems_folder>/workspace
   ../scripts/makesymlinks.sh
   ```

4. Edit `<gems_folder>/common/Makefile.simics_version`, set version to 3.0.
5. Edit `<gems_folder>/common/Makefile.common` and make following modifications:
   - set HOST_TYPE to amd64-linux
   - set CC to g++
   - set GEMS_ROOT to `<gems_folder>`
   - set SIMICS_ROOT to `<simics_folder>`
   - set SIMICS_INCLUDE_ROOT to `<simics_folder>/src/include`

6. Edit `<gems_folder>/ruby/module/Makefile` and make following modifications:

   - set GEMS_ROOT to `<gems_folder>`
   - set CC_VERSION to the gcc version on system
   - set HOST_TYPE to amd64-linux

7. Edit `<gems_folder>/opal/module/Makefile` and make following modifications:

   - set GEMS_ROOT to `<gems_folder>`
   - set CC_VERSION to the gcc version on system
   - set HOST_TYPE to amd64-linux

8. Edit `<gems_folder>/tourmaline/module/Makefile` and make following modifications:

   - set GEMS_ROOT to `<gems_folder>`
   - set CC_VERSION to the gcc version on system
   - set HOST_TYPE to amd64-linux

9. Build GEMS code:

```
cd <gems_folder>/ruby
make PROTOCOL=MESI_SCMP_bankdirectory_m DESTINATION=MESI_SCMP_bankdirectory_m
cd <gems_folder>/opal
make module DESTINATION=MESI_SCMP_bankdirectory_m
```

After GEMS code is built, it can be executed under the workspace folder (which was created at first step):

```
cd <gems_folder>/workspace/home/MESI_SCMP_bankdirectory_m
./simics <options>
```

8.2 DEVELOPMENT AND METHODOLOGY

8.2.1 Simics g-cache Module

The cache/memory model provided by Simics is mostly in its "g-cache" module. Source code of the module can be copied to current workspace by using the "–copy-device" option:

```
<simics_folder>/bin/workspace-setup --copy-device=g-cache
```

This will copy the g-cache module code into `<workspace_folder>/modules/g-cache`.

G-cache provides a simple cache model using LRU policy. In case of multiprocessors, g-cache includes a sample snoop-based MESI implementation. Although g-cache is not a detailed model, it can be a good start point due to its simpleness. G-cache is also much faster than GEMS, making it very helpful for quick profiling or collecting long memory traces for further study. For example, the traces I used in my memory controller works were collected with Simics and g-cache.

The source code of g-cache is easy to understand. Most of the cache operations are handled in functions defined in `gc-specialize.c`. For example, cache read and write operations are handled by function `handle_read()` and `handle_write()` in the file. Researcher can also define custom attributes in file `gc-attributes.c`.

I remark that g-cache does not maintain the cache data in its data structure (i.e., it only tracks tags). Hence if researcher wants to analyze the data being read or written in memory hierarchy (e.g., study the bit changes for endurance research), g-cache module needs to be extended to track the data. In my implementation, I found two Simics calls to be crucial for serving this purpose:

- `SIM_c_get_mem_op_value_buf()`

  In case of a memory write instruction, this function retrieves the operand, which is the actual data to be written.

- `gc_read_from_phys_mem()`

  This function retrieves the data from the "physical memory" of the simulated machine.

I used these two functions in g-cache to track data flow in memory hierarchy when collecting traces for the endurance and memory controller studies.

A limitation of using Simics g-cache module to collect memory traces is that it does not include the memory accesses caused by DMA operations. Therefore the collected trace will only include the memory accesses generated by user applications and operating system.

8.2.2 GEMS Memory Model

The cache/memory model in GEMS (called Ruby) is much more detailed (and complicated) than Simics g-cache.

GEMS provides a framework for cache coherence protocols (in `<gems_folder>/protocols`) under which researchers can define their own protocols. I found the interface to be difficult to understand and debug, so it may not be a good idea to write a new coherence protocol. Fortunately, GEMS provides lots of sample coherence protocols that have been tested. In the most cases, researcher can use one of the sample protocols in GEMS. For example, I used their MESI protocol (MESI_SCMP_bankdirectory) in my memory controller studies.

GEMS also provides a model of the memory controller with DDR2 interface. The implementation is written in C++. Majority of the model is implemented in two files, `MemoryControl.C` and `MemoryControl.h`, under `<gems_folder>/ruby/system` folder. The procedure of processing a memory request in GEMS is as follows.

- Memory requests from cores or last-level cache are enqueued by `enqueue()` function into the request queue (the `m_input_queue` member of the class).
- The bank queues are implemented as an array of queues, corresponding to the `m_bankQueues` member in the MemoryControl class.
- The key function of GEMS memory model is `MemoryControl::executeCycle()`, which executes a memory cycle. In this function, requests are dispatched from request queue (`m_input_queue`) to individual bank queues. Hence GEMS employs a simple first-come-first-serve (FCFS) scheduling policy, meaning that requests are dispatched in the order as they arrive. As I described previously, `MemoryControl::executeCycle()` issues up to one request per memory cycle. This is done by checking each bank's readiness (`MemoryControl::queueReady()`) and issuing request to a "ready" bank. If multiple banks are ready, `MemoryControl::executeCycle()` will issue the requests in round-robin fashion.

In my research, I modified the GEMS memory code to add my PCM model, my scheduling enhancements and power budgeting techniques.

In addition to execution-driven mode, I also found a way to run GEMS in trace-driven mode. This is achieved by making an infinite loop in `MemoryControl::executeCycle()`. Once simulator goes into `MemoryControl::executeCycle()`, it falls into my infinite loop which reads traces from file and execute the `executeCycle()` function without giving control back to Simics. Once the trace

is finished, I use `exit()` system call to break the loop and exit to command line. This method has the advantage of trace-driven simulation (fast, deterministic and platform-independent), and allows me to make use of the many data structures and functions provided by GEMS. I can still compile my code under GEMS using the same set of scripts, instead of writing all of them from scratch. I think this could be a helpful experience to other researchers who want to use Simics/GEMS in their works.

9.0 FUTURE DIRECTIONS AND CONCLUSION

In this chapter, I will discuss some future research directions and conclude the dissertation.

9.1 FUTURE DIRECTIONS FOR RESEARCH

Although the techniques I proposed in my research have made significant improvements on PCM memory, there are still some areas that are worth exploration in the future.

9.1.1 Multi-Level Cells

One future research direction is the impact of multi-level cells on my techniques. In my proposed techniques, I assumed single-level cells (SLC) in my PCM memory design. Although my techniques are independent from PCM cell technologies, applying multi-level cell (MLC) does introduce new research opportunities.

As previous work has revealed, a write operation in MLC is even more expensive than in SLC because it is an iterative write-and-verify process [36]. This would make existing latency mitigation schemes like write-cancelation [76] less effective. It also makes PCM write latency to be even more non-deterministic, because writing a single PCM cell may take any number of iterations. This opens a new research opportunity for memory scheduling. Considering a write request that consists of several rounds, each cell in a single round may take drastically different times to write. Conservatively using the time of the slowest cell write as the latency of the round is clearly not efficient.

One possible way to improve this is to group cell writes with similar latencies in one round, so that the total write latency can be reduced. This may be achieved through encoding techniques in the memory controller (e.g., predefine a set of encodings and select one of them that yields the

best result at runtime). Another possible improvement is to allow write drivers that finish early to advance to next round early. This may require some changes in bank design as well as scheduling schemes in the memory controller.

9.1.2 Power Demand Estimation

The Bit Level Power Budgeting (BPB) technique I proposed utilizes the information from Differential Write to estimate power demand of each write request. While it is much better than assuming every bit will be changed, it is still a little conservative in that it assumes the round with the most number of bit changes as the power demand of the entire write request, but different rounds in a single write request could still have very different power demands. So a possible improvement to BPB is to make a disparate power demand estimation for each round. However, this could increase the complexity of calculating $\hat{T}_{start}$ at runtime (which is the reason I choose the conservative assumption). An improved BPB design should therefore include techniques to reduce the runtime overhead when exploiting this opportunity of more accurate power demand estimation.

9.1.3 Memory Controller Architecture

With the advances of memory scheduling, the memory controller is becoming more and more complex. Memory schedulers like PAR-BS [63] are too complex to be implemented using ASIC. Hence these sophisticated memory schedulers are likely to be be implemented as microcontrollers running firmware. The usage of wear-leveling algorithms in PCM memory also requires a microcontroller on the memory module.

With this trend, I believe that some of the scheduling tasks that could be done locally would be off-loaded to the memory module. As I have proposed in my BPB technique, I chose a decoupled design for more flexibility, which is an example of this idea. Fang et al. also proposed a similar idea in [21]. This would require significant changes to the architecture of the memory controller, which could be a high-impact topic in the future.

9.1.4 Convergence of Memory and Storage

Another more general and farther consideration is the possible convergence of memory and storage using PCM. PCM is widely regarded as having some advantages of both memory and storage.

For example, it is byte-addressable and has comparable read speed to DRAM, and it is also non-volatile which is similar to storage devices like Flash or hard disk. Hence PCM is also referred to as a "storage-class memory" (SCM). This gives PCM unique opportunity to make memory and storage converge with each other, forming a single uniform address space for operating systems and applications. In order to benefit from this different, flat memory hierarchy, both the operating system and the programming model need to be changed.

9.2 CONCLUSION

With the growing demand for large capacity main memory and the shrinking of feature sizes, DRAM-based main memory is facing serious leakage and scalability issues. Phase Change Memory (PCM) has emerged as a promising alternative due to its high density, low leakage, byte-addressability and good scalability. However, the application of PCM memory still faces challenges caused by PCM's disadvantageous write operations.

This thesis describes my effort towards the successful application of Phase Change Memory. I proposed Differential Write at the circuit level to remove unnecessary bit changes in PCM writes. Along with a set of simple wear-leveling techniques, the lifetime of PCM memory was extended to 22 years on average. To address the throughput issue of PCM memory, I proposed memory scheduling enhancements (AWP and RAWP) for non-blocking PCM bank design. A fine-grained power budgeting technique (BPB) was then proposed to improve throughput under power budgets. Finally, I proposed my techniques to extend the QoS tuning range of high-priority applications when running on PCM memory. In addition to the experiment results, I also presented my experimental infrastructure and my visions of potential topics, which could be helpful for future research.

What I learned from my dissertation research is that throughput is still one of the biggest challenges to PCM memory. While there have been many wear-leveling techniques to extend PCM memory's lifetime with low overhead, no other significant work, to the best of my knowledge, has been done on improving its throughput. And although my memory scheduling enhancements can achieve 50% $\sim$ 60% of throughput improvement, the throughput gap between PCM and DRAM is still large. Consider a PCM memory with 8 logic banks and 1000ns write access latency, its theoretical peak write bandwidth would be 64B per write request $\times$ 1000000 write requests per second $\times$ 8 logic banks = 512MB/s. On the contrary, peak bandwidth supported by DDR2-800 is 6.4GB/s [31]. This indicates more than 12$\times$ gap between the throughput of PCM and DRAM. Non-blocking bank design and my memory scheduling enhancements may close this gap to about 7.6$\times$ (assuming a 60% average throughput improvement), but there is still much space for further improvement. This issue can get worse with the wide adoption of multi-level PCM cells, whose write operations are even slower.

Moreover, such throughput improvement efforts on PCM must also take power into consideration, because improving throughput requires increasing parallelism, and raising power budget of PCM memory is expensive. When a power budget exists, it is important to improve its utilization

so that more parallelism can be exploited without having to raise the power budget. I found that reducing power demand (number of bit writes) of each write request is an effective way to achieve this goal.

In summary, I believe that PCM memory's throughput issue cannot be simply addressed by individual schemes. Instead, it requires a comprehensive solution that consists of several collaborating techniques from different levels. The method I took in my dissertation research is an example in this direction: It combines circuit level techniques like Differential Write and non-blocking bank design with architectural techniques like AWP/RAWP and power budgeting. It will not be surprising that more techniques from different levels are incorporated into this effort in the future.

BIBLIOGRAPHY

[1] R. Annunziata, P. Zuliani, M. Borghi, G. De Sandre, L. Scotti, C. Prelini, M. Tosi, I. Tortorelli, and F. Pellizzer. Phase Change Memory Technology for Embedded Non Volatile Memory Applications for 90nm and Beyond. In *International Electron Devices Meeting (IEDM)*, 2009.

[2] M. Awasthi, M. Shevgoor, K. Sudan, R. Balasubramonian, B. Rajendran, and V. Srinivasan. Handling Resistance Drift in Phase Change Memory - Device, Circuit, Architecture, and System Solutions. In *Non-Volatile Memory Workshop (NVMW)*, 2011.

[3] F. Bedeschi, R. Fackenthal, C. Resta, E.M. Donze, M. Jagasivamani, E. Buda, F. Pellizzer, D. Chow, A. Cabrini, G.M.A. Calvi, R. Faravelli, A. Fantini, G. Torelli, Duane Mills, R. Gastaldi, and G. Casagrande. A Multi-Level-Cell Bipolar-Selected Phase-Change Memory. In *International Solid-State Circuits Conference (ISSCC)*, pages 428–429, 2008.

[4] S. Bock, B. Childers, R. Melhem, D. Mosse, and Youtao Zhang. Analyzing the Impact of Useless Write-Backs on the Endurance and Energy Consumption of PCM Main Memory. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2011.

[5] V. Bohossian, A. Jiang, and J. Bruck. Buffer Coding for Asymmetric Multi-Level Memory. In *IEEE International Symposium on Information Theory (ISIT)*, 2007.

[6] E. Bozorg-Grayeli, J.P. Reifenberg, M.A. Panzer, J.A. Rowlette, and K.E. Goodson. Temperature-Dependent Thermal Properties of Phase-Change Memory Electrode Materials. *IEEE Electron Device Letters*, 32(9), 2011.

[7] S. Braga, A. Cabrini, and G. Torelli. Experimental Analysis of Partial-SET State Stability in Phase-Change Memories. In *IEEE Transactions on Electron Devices*, 2010.

[8] A.M. Caulfield, A. De, J. Coburn, T.I. Mollow, R.K. Gupta, and S. Swanson. Moneta: A High-performance Storage Array Architecture for Next-generation, Non-volatile Memories. In *International Symposium on Microarchitecture (MICRO)*, 2010.

[9] F. Chen. SmartSaver: Turning Flash Drive Into a Disk Energy Saver for Mobile Computers. In *The International Symposium on Low Power Electronics and Design (ISLPED)*, pages 412–417, 2006.

[10] Y.C. Chen, C.T. Rettner, S. Raoux, G.W. Burr, S.H. Chen, R.M. Shelby, M. Salinga, W.P. Risk, T.D. Happ, G.M. McClelland, M. Breitwisch, A. Schrott, J.B. Philipp, M.H. Lee, R. Cheek, T. Nirschl, M. Lamorey, C.F. Chen, E. Joseph, S. Zaidi, B. Yee, H.L. Lung,

R. Bergmann, and C. Lam. Ultra-Thin Phase-Change Bridge Memory Device Using GeSb. In *International Electron Devices Meeting (IEDM)*, pages 777–780, December 2006.

[11] Y.-H. Chiu, Y.-B. Liao, M.-H. Chiang, C.-L. Lin, W.-C. Hsu, P.-C. Chiang, Y.-Y. Hsu, W.-H. Liu, S.-S. Sheu, K.-L. Su, M.-J. Kao, and M.-J. Tsai. Impact of Resistance Drift on Multi-level PCM Design. In *International Conference on IC Design and Technology (ICICDT)*, 2010.

[12] S. Cho and H. Lee. Flip-N-Write: a simple deterministic technique to improve PRAM write performance, energy and endurance. In *42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 347–357, 2009.

[13] S.L. Cho, J.H. Yi, Y.H. Ha, B.J. Kuh, C.M. Lee, J.H. Park, S.D. Nam, H. Horii, B.K. Cho, K.C. Ryoo, S.O. Park, H.S. Kim, U-In Chung, J.T. Moon, and B.I. Ryu. Highly Scalable On-axis Confined Cell Structure for High Density PRAM beyond 256Mb. In *Symposium on VLSI Technology Digest of Technical Papers (SVLSI)*, pages 96–97, 2005.

[14] W. Y. Cho, B.-H. Cho, B.-G. Choi, H.-R. Oh, S. Kang, K.-S. Kim, K.-H. Kim, D.-E. Kim, C.-K. Kwak, H.-G. Byun, Y. Hwang, S. Ahn, G.-H. Koh, G. Jeong, H. Jeong, and K. Kim. A 0.18-um 3.0-V 64-Mb Nonvolatile Phase-Transition Random Access Memory (PRAM). *IEEE Journal of Solid-State-Circuits*, 40(1):293–300, 2005.

[15] H. Chung, B. H. Jeong, B. Min, Y. Choi, B.-H. Cho, J. Shin, J. Kim, J. Sunwoo, J.-M. Park, Q. Wang, Y.-J. Lee, S. Cha, D. Kwon, S. Kim, S. Kim, Y. Rho, M.-H. Park, J. Kim, I. Song, S. Jun, J. Lee, K. Kim, K.-W. Lim, W.-R. Chung, C. Choi, H. Cho, I. Shin, W. Jun, S. Hwang, K.-W. Song, K. Lee, S.-W. Chang, W.-Y. Cho, J.-H. Yoo, and Y.-H. Jun. A 58nm 1.8V 1Gb PRAM with 6.4MB/s Program BW. In *International Solid-State Circuits Conference (ISSCC)*, 2011.

[16] G.F. Close, U. Frey, J. Morrish, R. Jordan, S. Lewis, T. Maffitt, M. Breitwisch, C. Hagleitner, C. Lam, and E. Eleftheriou. A 512Mb Phase-Change Memory (PCM) in 90nm CMOS achieving 2b/cell. In *IEEE Symposium on VLSI Circuits, Digest of Technical Papers (VLSIT)*, 2011.

[17] The Standard Performance Evaluation Corporation. SPEC CPU2006. http://www.spec.org/cpu2006/.

[18] G. De Sandre, L. Bettini, A. Pirola, L. Marmonier, M. Pasotti, M. Borghi, P. Mattavelli, P. Zuliani, L. Scotti, G. Mastracchio, F. Bedeschi, R. Gastaldi, and R. Bez. A 4 Mb LV MOS-Selected Embedded Phase Change Memory in 90 nm Standard CMOS Technology. In *The IEEE Journal of Solid-State Circuits (JSSC)*, 2010.

[19] J. Dong, L. Zhang, Y. Han, Y. Wang, and X. Li. Wear Rate Leveling: Lifetime Enhancement of PRAM with Endurance Variation. In *Design Automation Conference (DAC)*, 2011.

[20] X. Dong, X. Wu, G. Sun, Y. Xie, H. Li, and Y. Chen. Circuit and Microarchitecture Evaluation of 3D Stacking Magnetic RAM (MRAM) as a Universal Memory Replacement. In *Design Automation Conference (DAC)*, pages 554–559, 2008.

[21] K. Fang, L. Chen, Z. Zhang, and Z. Zhu. Memory Architecture for Integrating Emerging Memory Technologies. In *The International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2011.

[22] A. Faraclas, N. Williams, A. Gokirmak, and H. Silva. Modeling of Set and Reset Operations of Phase-Change Memory Cells. In *IEEE Electron Device Letters*, 2011.

[23] A.P. Ferreira, S. Bock, B. Childers, R Melhem, and D. Mosse. Impact of Process Variation on Endurance Algorithms for Wear-Prone Memories. In *Design, Automation and Test in Europe (DATE)*, 2011.

[24] A.P. Ferreira, Miao Zhou, S. Bock, B. Childers, R. Melhem, and D. Mosse. Increasing PCM main memory lifetime. In *Design, Automation and Test in Europe (DATE)*, 2010.

[25] International Technology Roadmap for Semiconductors. International Technology Roadmap for Semiconductors. http://www.itrs.net/links/2007itrs/2007_chapters/2007_PIDS.pdf, 2007.

[26] A. Hay, K. Strauss, T. Sherwood, G. H. Loh, and D. Burger. Preventing PCM Banks from Seizing Too Much Power. In *International Symposium On Microarchitecture (MICRO)*, to appear, 2011.

[27] Y. Ho and et al. Nonvolatile Memristor Memory: Device Characteristics and Design Implications. In *The International Conference on Computer-Aided Design (ICCAD)*, 2009.

[28] J. Hu, C. J. Xue, W.-C. Tseng, M. Qiu Y. He, and E. H.-M. Sha. Reducing Write Activities on Non-volatile Memories in Embedded CMPs via Data Migration and Recomputation. In *Design Automation Conference (DAC)*, 2010.

[29] D. Ielmini, S. Lavizzari, D. Sharma, and A.L. Lacaita. Physical Interpretation, Modeling and Impact on Phase Change Memory (PCM) Reliability of Resistance Drift due to Chalcogenide Structural Relaxation. In *International Electron Devices Meeting (IEDM)*, 2007.

[30] Intel. Intel, STMicroelectronics Deliver Industry's First Phase Change Memory Prototypes. http://www.intel.com/pressroom/archive/releases/20080206corp.htm, 2008.

[31] JEDEC. PC2-6400/PC2-5300/PC2-4200/PC2-3200 Registered DIMM Design Specification. http://www.jedec.org/sites/default/files/docs/4_20_10R19A.pdf, 2010.

[32] A. Jiang, V. Bohossian, and J. Bruck. Floating Codes for Joint Information Storage in Write Asymmetric Memories. In *IEEE International Symposium on Information Theory (ISIT)*, 2007.

[33] L. Jiang, Y. Du, Y. Zhang, B.R. Childers, and J. Yang. LLS: Cooperative Integration of Wear-Leveling and Salvaging for PCM Main Memory. In *International Conference on Dependable Systems and Networks (DSN)*, 2011.

[34] L. Jiang, Y. Zhang, and J. Yang. Enhancing Phase Change Memory Lifetime through Fine-Grained Current Regulation and Voltage Upscaling. In *International Symposium on Low Power Electronics and Design (ISLPED)*, 2011.

[35] X. Jin, S. Jung, and Y. H. Song. Write-aware Buffer Management Policy for Performance and Durability Enhancement in NAND Flash Memory. *IEEE Transactions on Consumer Electronics*, pages 2393–2399, 2010.

[36] M. Joshi, W. Zhang, and T. Li. Mercury: A Fast and Energy-Efficient Multi-level Cell based Phase Change Memory System. In *The International Conference on High Performance Computer Architecture (HPCA)*, 2010.

[37] J.-Y. Jung and S. Cho. PRISM: Zooming in Persistent RAM Storage Behavior. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2011.

[38] M. Kanellos. IBM Changes Directions in Magnetic Memory. http://news.cnet.com/IBM-changes-directions-in-magnetic-memory/2100-1004_3-6203198.html, 2007.

[39] D.-H. Kang, J.-H. Lee, J.H. Kong, D. Ha, J. Yu, C.Y. Um, J.H. Park, F. Yeung, J.H. Kim, W.I. Park, Y.J. Jeon, M.K. Lee, Y.J. Song, J.H. Oh, G.T. Jeong, and H.S. Jeong. Two-bit Cell Operation in Diode-Switch Phase Change Memory Cells with 90nm Technology. In *IEEE Symposium on VLSI Technology Digest of Technical Papers*, pages 98–99, 2008.

[40] S. Kang, W. Y. Cho, B.-H. Cho, K.-J. Lee, C.-S. Lee, H.-R. Oh, B.-G. Choi, Q. Wang, H.-J. Kim, M.-H. Park, Y. H. Ro, S. Kim, C.-D. Ha, K.-S. Kim, Y.-R. Kim, D.-E. Kim, C.-K. Kwak, H.-G. Byun, G. Jeong, H. Jeong, K. Kim, and Y. Shin. A 0.1-μm 1.8-V 256-Mb Phase-Change Random Access Memory (PRAM) with 66-MHz Synchronous Burst-Read Operation. *IEEE Journal of Solid-State Circuits*, 42(1):210–218, January 2007.

[41] T. Kgil, S. D'Souza, A. Saidi, N. Binkert, R. Dreslinski, T. Mudge, S. Reinhardt, and K. Flautner. PicoServer: Using 3D Stacking Technology to Enable a Compact Energy Efficient Chip Multiprocessor. In *International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS)*, pages 117–128, 2006.

[42] T. Kgil, D. Roberts, and T. Mudge. Improving NAND Flash Based Disk Caches. In *The 35th International Symposium on Computer Architecture (ISCA)*, pages 327–338, 2008.

[43] K. Kim. Technology for sub-50nm DRAM and NAND Flash Manufacturing. In *International Electron Devices Meeting (IEDM)*, pages 323–326, 2005.

[44] S. Kim and H.-S.P. Wong. Generalized Phase Change Memory Scaling Rule Analysis. In *Non-Volatile Semiconductor Memory Workshop*, 2006.

[45] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter. Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior. In *The 43th IEEE/ACM International Symposium On Microarchitecture (MICRO)*, 2010.

[46] J. Kong and H. Zhou. Improving Privacy and Lifetime of PCM-based Main Memory. In *International Conference on Dependable Systems and Networks (DSN)*, 2010.

[47] S. Lai and T. Lowrey. OUM - A 180nm Nonvolatile Memory Cell Element Technology for Standalone and Embedded Applications. In *International Electron Devices Meeting (IEDM)*, pages 36.5.1–36.5.4, 2001.

[48] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting Phase Change Memory as a Scalable DRAM Alternative. In *The 36th International Symposium on Computer Architecture (ISCA)*, 2009.

[49] K.-J. Lee, B.-H. Cho, W.-Y. Cho, S. Kang, B.-G. Choi, H.-R. Oh, C.-S. Lee, H.-J. Kim, J.-M. Park, Q. W., M.-H. Park, Y.-H. Ro, J.-Y. Choi, K.-S. Kim, Y.-R. Kim, I.-C. Shin, K.-W. Lim, H.-K. Cho, C.-H. Choi, W.-R. Chung, D.-E. Kim, K.-S. Yu, G.-T. Jeong, H.-S. Jeong, C.-K. Kwak, C.-H. Kim, and K. Kim. A 90nm 1.8V 512Mb Diode-Switch PRAM with 266MB/s Read Throughput. In *International Solid-State Circuits Conference*, pages 472–616, 2007.

[50] K.-J. Lee, B.-H. Cho, W.-Y. Cho, S. Kang, B.-G. Choi, H.-R. Oh, C.-S. Lee, H.-J. Kim, J.-M. Park, Q. Wang, M.-H. Park, Y.-H. Ro, J.-Y. Choi, K.-S. Kim, Y.-R. Kim, I.-C. Shin, K.-W. Lim, H.-K. Cho, C.-H. Choi, W.-R. Chung, D.-E. Kim, K.-S. Yu, G.-T. Jeong, H.-S. Jeong, C.-K. Kwak, C.-H. Kim, and K. Kim. A 90nm 1.8V 512Mb Diode-Switch PRAM with 266MB/s Read Throughput. *IEEE Journal of Solid-state Circuits*, 43(1), January 2008.

[51] C. Lefurgy, K. Rajamani, F. Rawson, W. Felter, M. Kistler, and T.W. Keller. Energy Management for Commercial Servers. *IEEE Computer*, 36(12):39–48, 2003.

[52] D. L. Lewis and H.-H. S. Lee. Architectural Evaluation of 3D Stacked RRAM Caches. In *IEEE International 3D System Integration Conference (3DIC)*, 2009.

[53] J. Li, Chao-I Wu, S.C. Lewis, J. Morrish, Tien-Yen Wang, R. Jordan, T. Maffitt, M. Breitwisch, A. Schrott, R. Cheek, H.-L. Lung, and C. Lam. A Novel Reconfigurable Sensing Scheme for Variable Level Storage in Phase Change Memory. In *International Memory Workshop (IMW)*, 2011.

[54] L. Li, K. Lu, B. Rajendran, T.D. Happ, H.-L. Lung, C. Lam, and M. Chan. Driving Device Comparison for Phase-Change Memory. In *IEEE Transactions on Electron Devices*, 2011.

[55] J. Liang, R.G.D. Jeyasingh, H.-Y. Chen, and H.-S.P. Wong. A 1.4uA Reset Current Phase Change Memory Cell with Integrated Carbon Nanotube Electrodes for Cross-Point Memory Application. In *IEEE Symposium on VLSI Technology Digest of Technical Papers (VLSIT)*, 2011.

[56] P.S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, 2002.

[57] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *Computer Architecture News (CAN)*, 2005.

[58] J. D. McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA)*, 1995.

[59] MICRON. 2Gb DDR2 SDRAM Component: MT47H256M8HG-25. http://www.micron.com/products/partdetail?part=MT47H256M8HG-25.

[60] M.G. Mohammad, L. Terkawi, and M. Albasman. Phase Change Memory Faults. In *The 19th International Conference on VLSI Design*, pages 108–112, 2006.

[61] W. Mueller, G. Aichmayr, W. Bergner, E. Erben, T. Hecht, C. Kapteyn, A. Kersch, S. Kudelka, F. Lau, J. Luetzen, A. Orth, J. Nuetzel, T. Schloesser, A. Scholz, U. Schroeder, A. Sieck, A. Spitzer, M. Strasser, P.-F. Wang, S. Wege, and R. Weis. Challenges for the DRAM Cell Scaling to 40nm. In *International Electron Devices Meeting (IEDM)*, pages 336–339, 2005.

[62] O. Mutlu and T. Moscibroda. Stall-time Fair Memory Access Scheduling for Chip Multiprocessors. In *The 40th IEEE/ACM International Symposium On Microarchitecture (MICRO)*, 2007.

[63] O. Mutlu and T. Moscibroda. Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems. In *35th International Symposium on Computer Architecture*, pages 63–74, 2008.

[64] R. G. Neale, D. L. Nelson, and G. E. Moore. Nonvolatile and Reprogramable, the Read-Mostly Memory is Here. *Electronics*, 1970.

[65] K.J. Nesbit, N. Aggarwal, J. Laudon, and J.E. Smith. Fair queuing Memory Systems. In *The 39th IEEE/ACM International Symposium On Microarchitecture (MICRO)*, 2006.

[66] Numonyx. The Basics of Phase Change Memory Technology.

[67] Numonyx Chief Technology Office. Phase change memory and its impacts on memory hierarchy, Carnegie Mellon, September, 2009. http://www.pdl.cmu.edu/SDI/2009/slides/Numonyx.pdf.

[68] J. H. Oh and et al. Full Integration of Highly Manufacturable 512Mb PRAM Based on 90nm Technology. In *International Electron Devices Meeting (IEDM)*, pages 49–52, 2006.

[69] J.H. Oh, J.H. Park, Y.S. Lim, H.S. Lim, Y.T. Oh, J.S. Kim, J.M. Shin, Y.J. Song, K.C. Ryoo, D.W. Lim, S.S. Park, J.I. Kim, J.H. Kim, J. Yu, F. Yeung, C.W. Jeong, J.H. Kong, D.H. Kang, G.H. Koh, G.T. Jeong, H.S. Jeong, and K. Kim. Enhanced Write Performance of a 64-Mb Phase-Change Random Access Memory. *IEEE Journal of Solid-State-Circuits*, 41(1):122–126, 2006.

[70] S. T. On, H. Hu, Y. Li, and J. Xu. Lazy-Update B+-Tree for Flash Devices. In *International Conference on Mobile Data Management: Systems, Services and Middleware (MDM)*, 2009.

[71] S. R. Ovshinsky. Reversible Electrical Switching Phenomenon in Disordered Structures. *Physics Review Letters*, 1968.

[72] N. Papandreou, H. Pozidis, A. Pantazi, A. Sebastian, M. Breitwisch, C. Lam, and E. Eleftheriou. Programming Algorithms for Multilevel Phase Change Memory. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, 2011.

[73] H. Park, S. Yoo, and S. Lee. Power Management of Hybrid DRAM/PRAM-Based Main Memory. In *Design Automation Conference (DAC)*, 2011.

[74] A. Pirovano, A.L. Lacaita, A. Benvenuti, F. Pellizzer, S. Hudgens, and R. Bez. Scaling Analysis of Phase-Change Memory Technology. In *International Electron Devices Meeting (IEDM)*, pages 29.6.1–29.6.4, 2003.

[75] M. K. Qureshi, M. M. Franceschini, L. A. Lastras-Montano, and J. P. Karidis. Morphable Memory System: A Robust Architecture for Exploiting Multi-Level Phase Change Memories. In *The International Symposium on Computer Architecture (ISCA)*, 2010.

[76] M. K. Qureshi, M.M. Franceschini, and L.A. Lastras-Montano. Improving read performance of phase change memories via write cancellation and write pausing. In *The IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 163–173, 2010.

[77] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali. Enhancing Lifetime and Security of Phase Change Memories via Start-Gap Wear Leveling. In *The 42th International Symposium on Microarchitecture (MICRO)*, 2009.

[78] M. K. Qureshi and Y. N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *The 39th IEEE/ACM International Symposium On Microarchitecture (MICRO)*, 2006.

[79] M. K. Qureshi, A. Seznec, L.A. Lastras, and M.M. Franceschini. Practical and Secure PCM Systems by Online Detection of Malicious Write Streams. In *The International Conference on High Performance Computer Architecture (HPCA)*, 2011.

[80] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable High Performance Main Memory System Using Phase-Change Memory Technology. In *The 36th International Symposium on Computer Architecture (ISCA)*, 2009.

[81] N. Rafique, W.-T. Lim, and M. Thottethodi. Effective Management of DRAM Bandwidth in Multicore Processors. In *The International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2007.

[82] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y.-C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H.-L. Lung, and C. H. Lam. Phase-Change Random Access Memory: A Scalable Technology. *IBM Journal of Research and Development*, 52(4/5):465–479, 2008.

[83] S. Rixner. Memory Controller Optimizations for Web Servers. In *International Symposium on Microarchitecture (MICRO)*, 2004.

[84] S. Rixner, W.J. Dally, U.J. Kapasi, P. Mattson, and J.D. Owens. Memory Access Scheduling. In *International Symposium on Computer Architecture (ISCA)*, 2000.

[85] S. Schechter, G. H. Loh, K. Straus, and D. Burger. Use ecp, not ecc, for hard failures in resistive memories. In *International Symposium on Computer Architecture (ISCA)*, 2010.

[86] N. H. Seong, D. H. Woo, and H.-H. Lee. SAFER: Stuck-At-Fault Error Recovery for Memories. In *The 43th IEEE/ACM International Symposium On Microarchitecture (MICRO)*, 2010.

[87] L. Shi, C. Xue, and X. Zhou. Cooperating Write Buffer Cache and Virtual Memory Management for Flash Memory Based Systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2011.

[88] J. P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *Computer Architecture News*, 20(1):5–44, 1992.

[89] A. Snavely and D. Tullsen. Symbiotic Jobscheduling for a Simultaneous Multithreading Processor. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2000.

[90] N. H. Soeng and et al. Security Refresh: Prevent Malicious Wear-out and Increase Durability for Phase-Change Memory with Dynamically Randomized Address Mapping. In *The 37th International Symposium on Computer Architecture (ISCA)*, 2010.

[91] G. Sun, C. J. Hughes, C. Kim, J. Zhao, C. Xu, Y. Xie, and Y.-K. Chen. Moguls: a Model to Explore the Memory Hierarchy for Bandwidth Improvements. In *International Symposium on Computer Architecture (ISCA)*, 2011.

[92] F. Tabrizi. The Future of Scalable STT-RAM as a Universal Embedded Memory. *Embedded.com*, 2007.

[93] S. Thoziyoor, J.H. Ahn, M. Monchiero, J.B. Brockman, and N.P. Jouppi. A Comprehensive Memory Modeling Tool and its Application to the Design and Analysis of Future Memory Hierarchies. In *The 35th International Symposium on Computer Architecture (ISCA)*, pages 51–62, 2008.

[94] TOSHIBA. TOSHIBA develops 32MB read while write NOR memory devices. http://www.toshiba.com/taec/news/press_releases/2000/to-077.jsp.

[95] C. Villa, D. Mills, G. Barkley, H. Giduturi, S. Schippers, and D. Vimercati. A 45nm 1Gb 1.8V Phase-Change Memory. In *International Solid-State Circuits Conference (ISSCC)*, February, 2010.

[96] C.-Y. Wen, J. Li, S. Kim, M. Breitwisch, C. Lam, J. Paramesh, and L.T. Pileggi. A Non-volatile Look-Up Table Design Using PCM (Phase-Change Memory) Cells. In *IEEE Symposium on VLSI Circuits Digest of Technical Papers (VLSIC)*, 2011.

[97] M. Wu and et al. eNVy: A Nonvolatile Main Memory Storage System. In *International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS)*, pages 86–97, 1994.

[98] X. Wu, J. Li, L. Zhang, E. Speight, and Y. Xie. Hybrid Cache Architecture with Disparate Memory Technologies. In *The 36th International Symposium on Computer Architecture (ISCA)*, 2009.

[99] F. Yeung, S.-J. Ahn, Y.-N. Hwang, C.-W. Jeong, Y.-J. Song, S.-Y. Lee, S.-H. Lee, K.-C. Ryoo, J.-H. Park, J.-M. Shin, W.-C. Jeong, Y.-T. Kim, G.-H. Koh, G.-T. Jeong, H.-S. Jeong, and K. Kim. $Ge_2Sb_2Te_5$ Confined Structures and Integration of 64Mb Phase-Change Random Access Memory. In *Japanese Journal of Applied Physics*, pages 2691–2695, 2005.

[100] D. H. Yoon, N. Muralimanohar, J. Chang, P. Ranganathan, N. P. Jouppi, and M. Erez. FREE-p: Protecting Non-Volatile Memory against both Hard and Soft Errors. In *HPCA*, 2011.

[101] W. Zhang and T. Li. Helmet: A Resistance Drift Resilient Architecture for Multi-level Cell Phase Change Memory System. In *Dependable Systems and Networks (DSN)*, 2011.

[102] P. Zhou, Y. Du, Y. Zhang, and J. Yang. Fine-Grained QoS Scheduling for PCM-based Main Memory Systems. In *International Parallel & Distributed Processing Symposium (IPDPS)*, 2010.

[103] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. A Durable and Energy Efficient Main Memory Using Phase Change Memory Technology. In *The 36th International Symposium on Computer Architecture (ISCA)*, pages 14–23, June 2009.

[104] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. Energy Reduction for STT-RAM Using Early Write Termination. In *International Conference on Computer-Aided Design (ICCAD)*, November 2009.

[105] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. Throughput Enhancement for Phase Change Memories. Technical report, University of Pittsburgh, 2011.